# Evaluating the Impact of Scaffolding and Visualizations for Mutation Testing Exercises in Software Engineering Education

Hannah Potter
hkpotter@cs.washington.edu
University of Washington
Seattle, USA

Ana C. R. Paiva
apaiva@fe.up.pt
INESC TEC, Faculty of Engineering,
University of Porto
Porto, Portugal

Domenico Amalfitano
domenico.amalfitano@unina.it
University of Naples Federico II
Naples, Italy

Anna Rita Fasolino
fasolino@unina.it
University of Naples Federico II
Naples, Italy

Porfirio Tramontana
ptramont@unina.it
University of Naples Federico II
Naples, Italy

René Just
rjust@cs.washington.edu
University of Washington
Seattle, USA

## Abstract

Mutation testing is an effective testing technique for improving how well a test suite can detect small changes to a program under test. This testing technique is seeing increased industry adoption. This paper aims to study the use of mutation testing in an educational setting and understand students' technical and conceptual challenges in applying mutation testing concepts. We report on two case studies of incorporating mutation testing into software engineering curricula.

The Scaffolding Study explores the impact of using different mutation analysis tools directly or indirectly via a uniform interface provided by an educational infrastructure. We observe that scaffolding (indirect tool use) improved the consistency of student performance for those using the same mutation analysis tool on the same code as well as helping students perform more effective mutation testing.

The Visualization Study explores the impact of different forms of output of a mutation analysis tool. Specifically, it assesses to what extent visualizations support students in reasoning about mutants and writing tests to detect them. We observe that like scaffolding, visualizations helped students perform more effective mutation testing, with lower-performing students seeing a boost in particular.

We further explore challenges around automatic assessment of mutation testing exercises. For example, we observe that even with assignment scaffolding, 18-21% of student submissions required manual modifications to successfully execute.

## CCS Concepts

• **Social and professional topics → Software engineering education**.

## Keywords

Mutation Testing, Software Testing Tools, Learning by Doing

**ACM Reference Format:**
Hannah Potter, Ana C. R. Paiva, Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and René Just. 2025. Evaluating the Impact of Scaffolding and Visualizations for Mutation Testing Exercises in Software Engineering Education. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3696630.3727240

## 1 Introduction

Software testing improves confidence in the correctness of a program, with various approaches to measuring test quality. Mutation analysis [41, 42] is one such approach seeing industry adoption [4, 32, 37], indicating a need for software engineers to learn mutation analysis and testing techniques. Software engineering and computer science students face challenges learning new and complex testing techniques [5, 14]. Hence, we seek to better understand the challenges and infrastructure implications for students to comprehend and perform mutation testing in an educational setting.

The use of mutation analysis and testing in educational contexts has been studied, including the development of a mutation testing educational module [26] and a gamification teaching approach [13], as well as using mutation analysis to obtain feedback about the quality of test suites produced by students [9, 23, 38]. While these studies provide valuable insights into facets of the use of mutation analysis and testing in educational contexts, to our knowledge, no prior work has investigated the advantages and drawbacks of using different mutation analysis tools from the perspective of both teachers and students. Identifying the challenges experienced by students when they struggle to learn a new testing technique and proposing pedagogical approaches for overcoming them is necessary [5].

We focused on technical and conceptual challenges students faced performing mutation testing in a classroom setting. We observed while teaching mutation testing exercises that technical challenges centered around difficulties configuring, running, and interpreting output of different mutation analysis tools. Conceptual challenges centered around difficulty understanding concepts relevant to mutation testing (e.g., state propagation). To address these challenges, we investigated different uses of scaffolding (direct vs. indirect use) of mutation analysis tools and visualization (as opposed to a textual representation) of mutation analysis reports and assess the effectiveness, consistency, and quality of student-produced test cases and short-answer responses.

The **Scaffolding Study** (Sec. 4) assesses to what extent mutation tool scaffolding supports student learning. In the Direct Tool Use Phase, students directly used different mutation analysis tools; in the Scaffolding Phase, students indirectly used different mutation analysis tools via a uniform interface in an educational infrastructure. Comparing the effectiveness and consistency of student submissions between both phases, we observe that students had more consistent performance with scaffolding. Students also had less mutant detection overlap w.r.t. developer tests, which may be an indication of a more accurate application of mutation testing.

The **Visualization Study** (Sec. 5) assesses to what extent additional visualization of mutation analysis outputs supports student learning. In the Textual Reports Phase, students use textual reports of mutation analysis output. In the Visual Reports Phase, students additionally have access to visualizations of this output. We observe that the visualizations boosted the accurate execution of mutation testing goals for lower-performing students and that students achieved similar mutation scores with less overlap w.r.t. provided starter tests.

Considering both studies, we discuss challenges related to automatic assessment (Sec. 6). We observe that even when provided a common assignment infrastructure, approximately a fifth of student submissions required manual modifications to be executable.

In summary, the main contributions of this work are:

- a study of student use of individual mutation analysis tools vs. a scaffolded assignment infrastructure
- a study of the impact of textual vs. visual mutation analysis reports on student performance and understanding
- identification of challenges around automated assessment of mutation testing exercises

## 2 Background

There are several measures used to evaluate the quality of a test suite. One common measure is *code coverage*, which assesses the percentage of statements, conditions, etc. that are executed by a test suite. While code coverage is a relatively cheap value to measure, a test suite having high code coverage does not necessarily indicate fault detection capability. For instance, a test suite which executes every line in a program, but has no assertions would have 100% line coverage but would not discover program faults other than those related to program exit statuses.

*Mutation analysis* provides a stronger measure of the quality of a test suite than code coverage [41, 42]. Mutation analysis makes small changes, *mutations*, to a program to create many different *mutant* programs. These changes are made by applying *mutation operators* (e.g., changing an `==` to a `!=`). If a test suite can observe the difference between the original program and a mutant (e.g., the test suite passes on the original program and fails on the mutant), that mutant is *detected*. Otherwise, the mutant is *undetected*. The *mutation score* gives the percentage of detected over the total number of generated mutants. Mutants and mutation scores are coupled to real faults and correlated with real fault detection [15]. It is possible for the original program and the mutant to be semantically equivalent (e.g., removing a statement in dead code). Such a mutant is called an *equivalent mutant* and is not detectable.

A mutant must infect state to be detectable. For instance, if the condition `a <= 0` is mutated to `a <= -1`, for the mutant to be detected, an `a` is needed such that `a <= 0` and `a <= -1` have different truth values (i.e. `a == 0`). However, *state infection* alone is not sufficient for a mutant to be detected; there must also be *state propagation*. The infected state must be propagated such that the difference between the original and mutant programs can be observed. For instance, if the previous condition is part of a larger expression such as `a <= 0 || b <= 0`, `a == 0` alone is not sufficient to observe the mutant. The additional condition that `b <= 0` is `false` must also be met. This reasoning must be extended through the rest of the program.

Mutation analysis provides a mutation score for a given test suite and set of mutants (i.e. the goal is to get a measure of how good a test suite is). *Mutation testing* is similar but is a process by which a developer tries to write a test that detects an undetected mutant (i.e. the goal is to add tests that detect mutants) [2, 22].

A mutant can be considered *productive* or *unproductive* [4, 31, 32]. A mutant is defined as productive if it is detectable and elicits an effective test or if it is equivalent, but its analysis and resolution improve code quality or knowledge [33]. Productiveness is a subjective measure as the effectiveness of a test, code quality, or improved knowledge of code can vary by developer.

## 3 Experimental Setup

This section outlines the methodology common to both the Scaffolding Study and the Visualization Study. In particular, this section describes the subject programs, mutation analysis tools, and measures of interest. Further aspects specific to each case study are presented in their respective sections (Sec. 4 and Sec. 5).

Both case studies are organized into two phases: the first observes challenges around mutation testing exercises and serves as a control; the second implements solutions for the observed challenges and repeats the same mutation testing exercise. Both studies were performed in common educational settings, and all data used was anonymized.

### 3.1 Subject Programs

Across the case studies, we use four subject programs (see Tab. 1).

**Triangle** is a self-contained Java program used for educational purposes. It takes as input three integers representing the side lengths of a triangle and outputs the classification of the triangle (equilateral, scalene, isosceles, or invalid). The release of the Major mutation framework [18, 19] includes the `Triangle` program as an example. We used `Triangle` in our case studies because it is a short program with an intuitive specification while still having non-trivial control flow.

**DateUtils** is part of the Lang project and has utility functions for using Java Calendar and Date objects. **HelpFormatter** is part of the Cli project and is a formatter for help messages for command line options. **JsonWriter** is part of the Gson project and has utility functions for writing JSON to a stream. Lang, Cli, and Gson are projects from Defects4J (v2.0.0), a popular benchmark in software engineering research [20]. We used these specific classes in the selected projects because they provide self-contained functionality and are amenable to straightforward unit testing (e.g., no need for mocking).

**Table 1: Selected subject programs.**

| Class | Project | LOC | Defects4J Version |
|---|---|---|---|
| Triangle | Triangle | 52 | *NA* |
| DateUtils | Lang | 990 | Lang-53f |
| HelpFormatter | Cli | 1010 | Cli-32f |
| JsonWriter | Gson | 659 | Gson-15f |

## 3.2 Mutation Analysis Tools

Across the case studies, we use four different state-of-the-art mutation tools: Major [19, 21], PIT [8], Jumble [16], and Judy [25]. We chose these tools because (1) there has been successful application of these tools on real-world projects and (2) these tools differ across multiple dimensions, such as level of detail in generated outputs and mutant-generation approach (e.g., mutation of source vs. byte code) [2].

We distinguish between direct use and scaffolding of mutation analysis tools. Direct use refers to students using a tool directly (i.e., downloading and installing the tool and running it according to the tool documentation). Scaffolding refers to students using a tool through an exercise infrastructure. These infrastructures are detailed in each case study.

## 3.3 Measures of Interest

Across the case studies, we investigate student performance and understanding of mutation testing.

*3.3.1 Effectiveness and Consistency.* We investigated students' effectiveness and consistency on mutation testing exercises. Improving consistency of answers among students narrows the gap between the lower and higher performing students and eases grading in a classroom setting. We consider the following measures:

**Number of Tests Written** is the number of tests that students added to their given starter test suites. We expect more effective students to write more tests with scaffolding because they can spend more time writing tests than struggling with tool complexity. We also expect more effective students to write fewer tests with visualization because they will write more targeted tests (i.e. reduction of "guess and check" strategies in favor of more strategic testing).

**Line and Condition Coverage** is the percentage of lines and conditions executed by a test suite. We expect more effective students to have higher line and condition coverage.

**Mutation Score** is the percentage of detected over generated mutants for a given test suite. We expect more effective students to have higher mutation scores.

**Detected Mutants Overlap** compares the proportion of mutants that are detected by only student-written tests or only starter tests, by both student-written and starter tests, or by neither. Reducing the overlap between student-written tests and starter tests is indicative of more strategic and targeted mutation testing. We expect more effective students to have less overlap.

*3.3.2 Quality of Rationale on Equivalent and Productive Mutants.* We additionally investigated students' comprehension and understanding of mutation testing. In addition to writing tests, students provided rationale about mutant equivalence and productiveness.

**Table 2: Number of submissions for each class and tool for the Scaffolding Study.**

| Class | Tool | Number of Submissions | |
|---|---|---|---|
| | | Direct | Scaffolding |
| **DateUtils** | **Judy** | 2 | 4 |
| | **Jumble** | 4 | NA |
| | **Major** | 2 | 5 |
| | **PIT** | 3 | 5 |
| **HelpFormatter** | **Judy** | 2 | 3 |
| | **Jumble** | 3 | NA |
| | **Major** | 3 | 4 |
| | **PIT** | 2 | 4 |
| **JsonWriter** | **Judy** | 3 | 5 |
| | **Jumble** | 3 | NA |
| | **Major** | 3 | 5 |
| | **PIT** | 3 | 4 |

Two authors individually classified each submission. The two authors discussed where classifications differed. Each submission was classified as:

- **None**: No justification provided or no classification.
- **Unclear**: Cannot understand the rationale or rationale does not show signs of understanding the concept.
- **Generic**: Too generic; no meaningful application of concepts.
- **Sufficient**: Some meaningful application of concepts (even if fairly minimal).

## 4 Scaffolding

The Scaffolding Study focuses on addressing technical challenges students have with using mutation analysis tools.

## 4.1 Experimental Setup

The Scaffolding Study was performed over two consecutive editions of an in-person Software Testing Master's course, where students are primarily recent graduates. Theoretical classes are used to formally present the teaching topics, while lab classes are used to carry out small projects and explore different testing tools. Additionally, there is an end-of-semester lab project.

For the lab project, students are asked to use mutation testing to detect mutants that are undetected by developer tests for a given project(s). The high-level learning goals of this project are to learn about systematic unit testing and reason about test quality, using coverage and mutation analysis criteria.

We kept the core material of the mutation testing lab project the same over the two course editions but provided a scaffolded mutation testing environment in the second edition to assess its impact on observed technical challenges. To summarize the two course editions:

**Direct Tool Use Phase** Students directly used different mutation analysis tools for the lab project. Students were divided into 16 groups of two to three students and 1 group of one student. We assigned one tool with two Defects4J classes under test to each

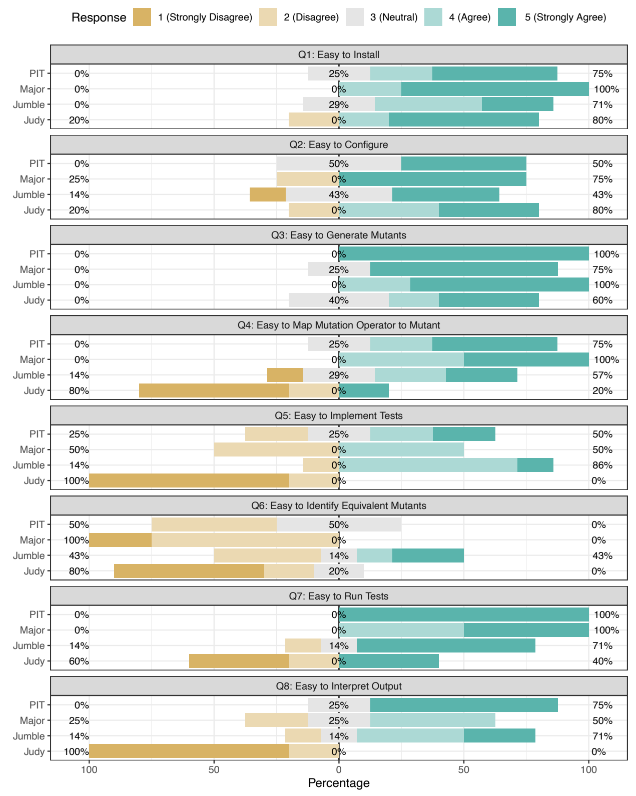


**Figure 1: Survey responses for the Direct Tool Use Phase.**

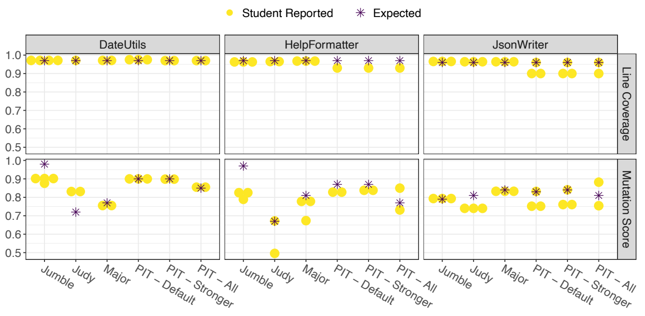


**Figure 2: Student reported and expected test quality measures for the provided starter tests for the Direct Tool Use Phase.**

group with two or more students and one tool with one class to the group with a single student. Tool and class assignments were random. This resulted in 17 total groups and 33 total class-tool submissions for this phase (Tab. 2).

**Scaffolding Phase** Students indirectly used the mutation analysis tools through a provided infrastructure for the lab project. Students worked individually and each student was randomly assigned one tool and class. We analyzed 39 class-tool submissions from this phase (Tab. 2).

## 4.2 Direct Tool Use Phase

We observed a few technical issues that students struggled with while working on the lab project during the Direct Tool Use Phase.

*4.2.1 Usability of Mutation Analysis Tools for Students.* Students were asked to answer a questionnaire we designed about the usability of their assigned mutation analysis tool. The Likert questions and results are summarized in Fig. 1. Judy was the tool that generated the most dissatisfaction among students. The most significant difficulties using Judy were related to implementing and running tests and interpreting outputs.

At the end of the questionnaire, students were given the opportunity to provide additional comments about the tool they used. We highlight a few themes for each tool:

- **Judy**: missing and unclear information in reports; difficulty understanding how to use the tool and outputs; lack of documentation; long execution time.
- **PIT**: missing and unclear information in reports; difficult installation and configuration process.
- **Jumble**: missing and unclear information in reports; lack of documentation.
- **Major**: missing detailed information in reports.

The observed variability in tool usability creates unwanted inequalities in an educational setting where some students may have more difficult tool assignments than others. From the instructor perspective, teaching how to use multiple different tools is difficult, especially given varying quality of tool documentation. Thus instructors may spend more time helping students overcome technical issues than clarifying the concepts for mutation testing.

*4.2.2 Analysis Results Reported by Students for Given Starter Tests.* Fig. 2 reports the expected values for line coverage and mutation score that should be obtained by executing the provided starter tests. The starter tests are developer-written tests that come with the Defects4J projects. Since PIT can be configured to generate different mutants, we considered all three configurations: Default, Stronger, and All mutation operators.

Fig. 2 also shows the values students reported after running the starter tests[1]. Students were more accurate for the more traditional test quality measure line coverage than mutation score. The discordance between the expected and student reported values for mutation score can occur from either differences in reporting of the number of generated mutants or the number of mutants detected by the starter tests[2].

There are two explanations suggested by the inconsistency in the students' reported results for the given starter tests. One is that different students are reporting the correct results for varying settings in their computing environments (e.g., which mutation operators to apply). These non-uniform environments make both manual and automatic assessment more difficult due to the complexity in

[1]We use the student reported numbers for detected and generated mutants to calculate the mutation score.

[2]Note that because mutation score is the number of detected mutants over the number of generated mutants, class-tool pairs with a smaller denominator will emphasize differences in mutation score more than those with larger denominators (e.g., being five mutants off from the expected when there are only 10 total mutants will affect the mutation score more than if there are 2000 mutants).

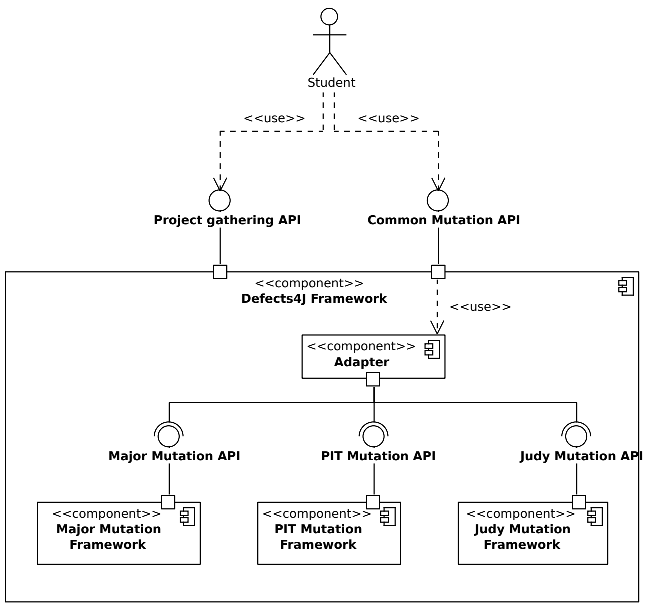


**Figure 3: UML Component Diagram showing the architectural design of the Defects4J extension.**

replicating the different settings used by students. The alternative explanation is that the students reported the incorrect results for their environment's configuration. Distinguishing between varying settings and erroneous answers for inconsistent results is difficult, making deciphering and assessing student work difficult.

## 4.3 Scaffolding Phase

Addressing the observed challenges in the Direct Tool Use Phase, we introduced a scaffolded environment in the Scaffolding Phase that aims to improve documentation and readability of generated reports while reducing mutation tool installation, configuration, and learning effort by providing a common environment to interact with each mutation tool.

Defects4J already provides an abstraction over the Major mutation analysis tool; users indirectly invoke Major through a command line interface. Thus, we extended Defects4J by adding two additional tools, PIT and Judy, to create a shared environment for all three tools. As an additional benefit, Defects4J already includes the software under test for the exercise. Jumble was excluded from this integration process because it does not generate the output artifacts necessary for automatic post-processing for the metrics in which we are interested.

The diagram in Fig. 3 describes the architecture of the Defects4J extension (D4JE). The extension provides a *Common Mutation API* with a shared set of commands for all the integrated mutation tools. This command line user interface abstracts away details of the syntax and output format of the individual mutation tools. The *Adapter* component binds the *Common Mutation API* to each of the *Mutation APIs* provided by the integrated mutation tools.

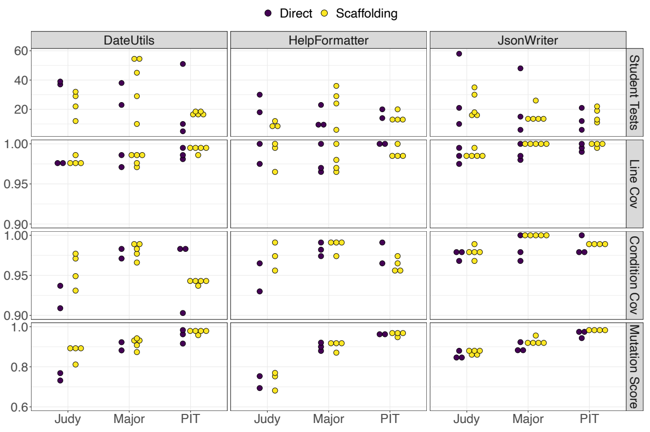


**Figure 4: Number of tests written by students as well as line coverage, condition coverage, and mutation scores of student and starter tests in the Scaffolding Study. For the mutation score for PIT, we show only the measures from the `Stronger` configuration. *Note that the y scales for each measure differ.***

## 4.4 Comparison and Discussion of Results

We analyzed submissions from the Direct Tool Use Phase and the Scaffolding Phase to assess student effectiveness, consistency, and quality of submissions.

*4.4.1 Effectiveness and Consistency.* We ran the student test suites from both the Direct Tool Use Phase and the Scaffolding Phase using the D4JE infrastructure[3] for each submissions' class-tool assignment. Since D4JE does not support Jumble, student submissions from the Direct Tool Use Phase that were assigned Jumble are excluded from this analysis. However, these Jumble submissions are included in the qualitative analysis in Sec. 4.4.2. Because students assigned to use PIT in the Direct Tool Use Phase used different PIT configurations (i.e. `Defaults`, `Stronger`, and `All`), we ran all PIT submissions for both phases against all PIT configurations. Students assigned PIT for the Scaffolding Phase used the `Stronger` configuration, which D4JE set.

Fig. 4 shows dot plots for the number of tests written by students, the line and condition coverage of student and developer tests, and the mutation score of student and developer tests[4]. Values are binned by 1/30 of the range of the data for each class-tool pair. We generally observe mixed results for different class-tool pairs. For students assigned to work with Judy, we observe an unexpected pattern, especially for DateUtils and HelpFormatter where students in the Scaffolding Phase wrote fewer tests but achieved similar or higher mutation scores. There may be an inverse relationship between the number of tests written and coverage and mutation scores, possibly from more systematic testing vs. "guess-and-check" strategies (see Sec. 3.3.1 for more discussion on expectations), but we leave investigation of this to future work with more data points.

[3]Minor modifications were made to the infrastructure to support gathering data.
[4]We compared all configurations of PIT as described above. We did not observe notably different patterns between `Default` and `Stronger`. For `All`, we see more students with higher scores in the Direct Tool Use Phase than for other configurations. This is not unexpected as students in this phase may have targeted mutants from the `All` configuration that would not have been presented to the students in Scaffolding Phase.

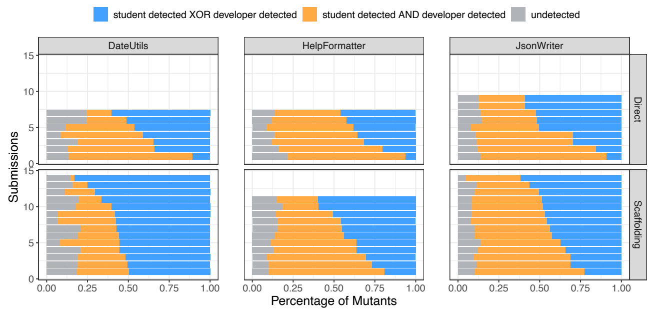


**Figure 5: Detected mutants overlap in the Scaffolding Study. Less mutant detection overlap (more blue) is better.**

We do generally observe more clustering of measures in the Scaffolding Phase, where more students reach the same or similar line coverage, mutation score, etc. This clustering indicates more consistency in student performance.

We further investigate student effectiveness and consistency by measuring the detected mutants overlap as described in Sec. 3.3.1. We calculated these measures by running the test suites with Major, regardless of the assigned tool. We chose to use one tool to provide a common set of mutants for each project to compare. We chose Major since it generates a kill matrix which provides information about which tests kill which mutants. We observe from Fig. 5 that generally students in the Scaffolding Phase had less overlap of mutants detected by both student and developer tests than in the Direct Tool Use Phase. These observations provide a signal that the students in the Scaffolding Phase performed more effective mutation testing, i.e., they understood better the detected and undetected mutants resulting from the execution of the starter tests.

#### 4.4.2 *Quality of Rationale on Equivalent and Productive Mutants.*

While scaffolding may not directly support understanding of equivalent and productive mutants, reducing technical difficulties may allow students to focus on learning and applying these concepts.

For each mutant analyzed, students were instructed to indicate and justify whether or not the mutant is equivalent. We observe an improvement in the reasoning about equivalence between the Direct Tool Use Phase and the Scaffolding Phase.

Students in the Direct Tool Use Phase were not asked to provide a justification for mutants they indicated were productive, so we cannot compare the quality of students' rationale. However, we do observe that most students in the Scaffolding Phase provided acceptable rationale for productive mutants.

## 5 Visualization

The Visualization Study focuses on addressing conceptual challenges students have with understanding mutation testing.

### 5.1 Experimental Setup

The Visualization Study involved two editions of the same undergraduate software engineering course. Additionally, this case study considers a similar graduate course as a reference point.

In the undergraduate course, students are typically majoring in computer science and are in the latter-half of their degree progression. The course is centered around a group project and includes in-class exercises focused on different software engineering topics, including software testing. Students typically work on in-class exercises in pairs, but may work individually. In-class exercises are scheduled during class time to allow students to work in pairs and to interact with the course staff. Questions and discussions of solutions are highly encouraged to prevent students from getting stuck and to encourage reflection on initial and refined solutions.

The graduate course's focus is on advanced topics in software engineering. The course focuses on static and dynamic program analyses, including software testing. The students are predominantly professional Master's students with software engineering related jobs who complete their degree as part-time students. The course focuses on in-class exercises, similar in structure to that of the undergraduate course.

Both courses involve an in-class exercise that covers mutation testing. The graduate exercise is more involved, with one part very similar to the undergraduate exercise. Students are given a small, self-contained program (`Triangle`). Students indirectly use the Major mutation framework. The code for the exercise is packaged such that students do not have to install the mutation testing software on their own; students simply clone a repository and run given scripts and build targets. Keeping the tools and program fixed aims to help students focus more on comprehension of mutation testing rather than struggling with technical issues such as tool installation. Students only need to add tests to a given test suite with the goal of reaching mutation adequacy (i.e. detect all non-equivalent mutants) for the `Triangle` program.

We kept the core material of the mutation testing exercise the same over two editions of the undergraduate course but provided additional tooling in the second edition to assess its impact on observed conceptual challenges. Additionally, the graduate course serves as a performance baseline with experienced software engineers. To summarize the three course editions:

**Baseline (graduate)** In a previous exercise, students were given a test suite with one example test for the `Triangle` program and asked to add tests to satisfy code coverage criteria. In a follow-up mutation testing exercise, students were asked to add tests to this test suite to satisfy mutation adequacy. We analyzed 24 submissions from this edition.

**Textual Reports Phase (undergraduate)** For the mutation testing exercise, students were provided a test suite that satisfied statement coverage and were asked to add tests to the given test suite to satisfy mutation adequacy. We analyzed 33 submissions from this course edition[5].

**Visual Reports Phase (undergraduate)** Students used the same setup and instructions as for the Textual Reports Phase, with the addition of tooling that aims to improve students' understanding of and reasoning about mutants. We analyzed 43 submissions from this course edition.

### 5.2 Textual Reports Phase

While we observed students working on the mutation testing exercise in the Textual Reports Phase, we noticed a few common concepts that students were distracted by or struggled with:

[5]One submission was discarded because the test source code was not submitted.

```
      /**
       * This static method does the actual classification of a triangle, given the lengths
       * of its three sides.
       */
      public static Type classify(int a, int b, int c) {
-         if (a <= 0 || b <= 0 || c <= 0) {
+         if (a <= -1 || b <= 0 || c <= 0) {
              return Type.INVALID;
          }
```

**Figure 6: An example mutant from the Visualization Study: the mutant changes a 0 to a -1 in an if-statement condition.**

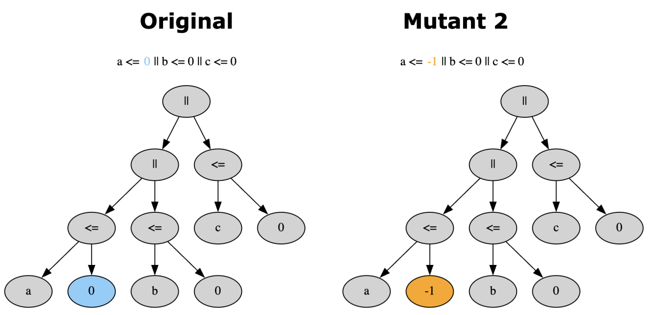
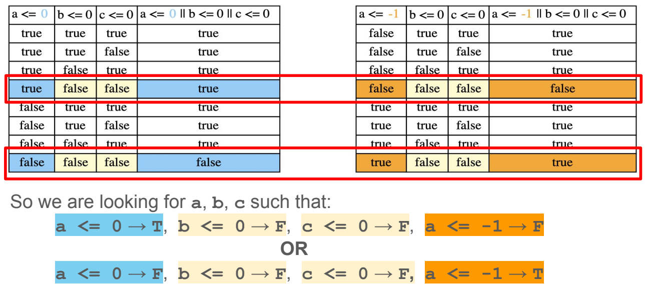


**Figure 7: Textual and AST diff of an example mutant from the Visual Reports Phase.**



| a <= 0 | b <= 0 | c <= 0 | a <= 0 \|\| b <= 0 \|\| c <= 0 |
|---|---|---|---|
| true | true | true | true |
| true | true | false | true |
| true | false | true | true |
| true | false | false | true |
| false | true | true | true |
| false | true | false | true |
| false | false | true | true |
| false | false | false | false |

| a <= -1 | b <= 0 | c <= 0 | a <= -1 \|\| b <= 0 \|\| c <= 0 |
|---|---|---|---|
| false | true | true | true |
| false | true | false | true |
| false | false | true | true |
| false | false | false | false |
| true | true | true | true |
| true | true | false | true |
| true | false | true | true |
| true | false | false | true |

So we are looking for `a`, `b`, `c` such that:

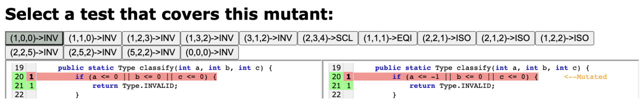
`a <= 0 → T`, `b <= 0 → F`, `c <= 0 → F`, `a <= -1 → F`

OR

`a <= 0 → F`, `b <= 0 → F`, `c <= 0 → F`, `a <= -1 → T`

**Figure 8: Example truth tables from the Visual Reports Phase, including annotations (shown in red) and an explanation (shown below the tables) of how to interpret the tables which was provided in tool documentation presented to students.**

- **Operator precedence and associativity**: Mutating operators can create less common combinations of operators that may be tricky to reason about. While the operator precedence and associativity is non-ambiguous, students were not always confident with combinations of operators with which they are less familiar. Being unsure of operator precedence and associativity can make it difficult to reason about state infection and propagation, thus making it more difficult to write tests that detect a mutant.
- **State infection and propagation**: Students sometimes had difficulty understanding why a test they wrote did not detect a targeted mutant due to challenges related to reasoning about state propagation. This included when, for the same input, the original and mutant programs have different execution paths but return the same value.



**Select a test that covers this mutant:**

(1,0,0)->INV (1,1,0)->INV (1,2,3)->INV (1,3,2)->INV (3,1,2)->INV (2,3,4)->SCL (1,1,1)->EQI (2,2,1)->ISO (2,1,2)->ISO (1,2,2)->ISO (2,2,5)->INV (2,5,2)->INV (5,2,2)->INV (0,0,0)->INV

```
       public static Type classify(int a, int b, int c) {
1          if (a <= 0 || b <= 0 || c <= 0) {
1              return Type.INVALID;
           }
```

```
       public static Type classify(int a, int b, int c) {
1          if (a <= -1 || b <= 0 || c <= 0) {       <--Mutated
1              return Type.INVALID;
           }
```

**Figure 9: Tests that cover an example mutant with coverage reports from the Visual Reports Phase (view cropped).**

## 5.3 Visual Reports Phase

To address the observed challenges in the Textual Reports Phase, we introduced additional tooling in the Visual Reports Phase. To reduce distraction from and improve comprehension of the core learning goals around mutation testing for the exercise, we created a visualization for the otherwise textual mutation analysis report.

The mutant visualization report is a web-based tool that aims to help students visualize and reason about mutants. This tool is not dependent on the mutation framework being used or a student's setup such as their preferred IDE. Students simply run a provided script which generates the report which can be viewed in a web-browser. The report consists of two main subsections for each undetected mutant: static information about the mutant and dynamic information about the tests that covered the mutant. For example, if a student is trying to write a test that detects the mutant shown in Fig. 6, they will see the following information to help overcome the observed challenges described in section 5.2:

- Operator precedence and associativity: Fig. 7 shows the **textual and AST diff** for a selected mutant. Color highlighting draws attention to the mutation. The diffs emphasize the precedence and associativity of the operators around a given mutation. This is static information about operator precedence and associativity for a selected mutant.
- State infection and propagation: Fig. 8 shows the **truth tables** for a selected mutant. The same color highlighting used for the textual and AST diffs is used to help associate related information. Highlighted rows extending across both tables give truth assignments that will result in both state infection and propagation to the parent expression. Yellow cells are common truth assignments between both tables. This is static information about state infection and propagation for a selected mutant. The top of Fig. 9 shows the test inputs and outputs that cover, but do not detect, the selected mutant. The bottom of Fig. 9 shows the **Cobertura code coverage reports** [7] for the selected test on the original program (left) and the mutant program (right). The code coverage reports give both line coverage and condition coverage information. This is dynamic information about state infection and propagation for a selected mutant.

*5.3.1 Experience Using the Mutant Visualization.* Students were asked to optionally complete an exit survey we designed asking about the overall experience of using the visualization tool and the usefulness of individual features. Based on interactions with students and the exit survey, we made the following observations. First, students reported that analysis and visualization speed was the predominant factor for their learning experience. While the visualization itself may be useful, it needs to have no noticeable latency between adding tests and viewing the reports to allow the

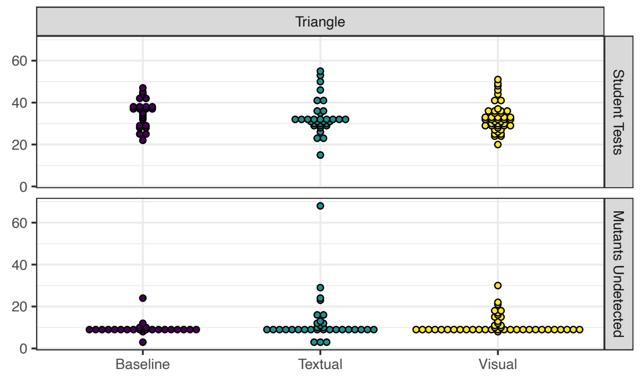


**Figure 10: Number of tests written by students and the number of undetected mutants (lower is better) in the Visualization Study. Since all submissions were using the same mutants on the same subject, we report on the number of undetected mutants instead of the mutation score.**

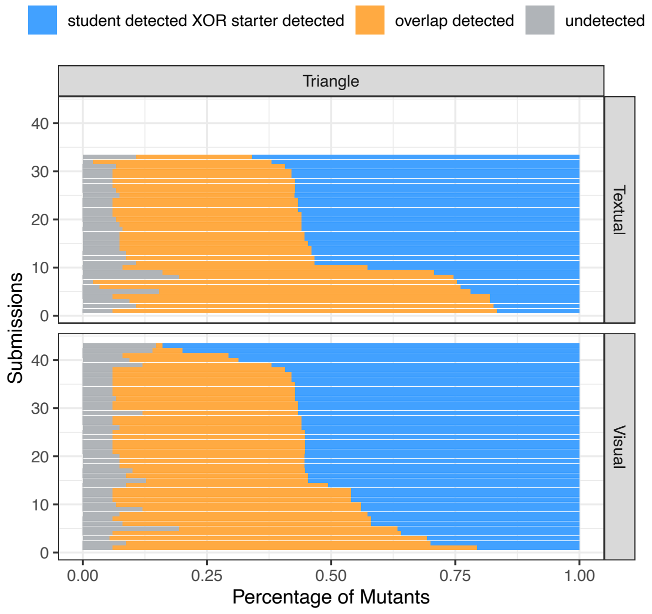


**Figure 11: Detected mutants overlap in the Visualization Study. Less mutant detection overlap (more blue) is better.**

students to focus on the exercise. Second, we observed that even with the provided assignment scaffolding, some students struggled setting up the exercise or made unexpected changes that broke the setup. Moving the assignment to be run entirely through a web-hosted GUI may allow students to focus more on mutation testing concepts instead of struggling with using the command line. Third, students sometimes looked at stale reports, which caused confusion. This was due either to students not understanding the need to regenerate the visualization after adding tests or a bug that was only found and fixed after the in-class work time. An automatic update on test file save could solve this problem. Fourth, we observed that students felt they had information overload with the reports. Allowing students to show/hide subsections of the visualization may ease this frustration. Additional exploration is needed to understand what information is most valuable to present as well as the most effective way to present the information.

## 5.4 Comparison and Discussion of Results

We again compared the Textual Reports Phase and the Visual Reports Phase to one another and to the Baseline to assess student effectiveness, consistency, and quality of submissions.

*5.4.1 Effectiveness and Consistency.* Fig. 10 shows the results for the number of tests written and the effectiveness of these tests in terms of the number of undetected mutants. All submissions except two achieved 100% line and condition coverage (one in the Baseline and one in the Textual Reports Phase).

We make three key observations on the effectiveness and consistency of student tests. First, the median number of tests is the same for Visual Reports Phase and the Textual Reports Phase (32.0). Second, the tests produced in the Textual Reports Phase and the Visual Reports Phase show overall similar effectiveness (median of 9.0 undetected mutants for both). Third, the variance for code coverage and the number of undetected mutants (and hence also the mutation score) is lower in the Visual Reports Phase. The standard deviations between the Textual Reports Phase and the Visual Reports Phase, respectively, for line coverage were 0.03 and 0.00, for condition coverage were 0.02 and 0.00, and for undetected mutants were 11.36 and 4.60. Additionally, the minimum code coverage is higher (0.84 vs. 1.00 for line coverage and 0.88 vs. 1.00 for condition coverage) and the maximum number of undetected mutants is lower (68 vs. 30) in the Visual Reports Phase versus the Textual Reports Phase, respectively, suggesting that the visualization may have particularly helped lower-performing students.

As in Sec. 4.4, we further investigate student effectiveness and consistency by calculating the detected mutants overlap. We observe from Fig. 11 that generally students in the Visual Reports Phase had less overlap of mutants detected by both student and starter tests than in the Textual Reports Phase. The median mutation score of the student tests without considering the starter tests, which is calculated by removing tests that are equivalent to the starter tests and calculating the mutation score from the remaining tests, is similar in Visual Reports Phase (0.53) versus Textual Reports Phase (0.52). We can conclude that the students in the Visual Reports Phase performed more effective mutation testing by achieving similar mutation scores while understanding and targeting mutants not detected by the starter tests.

While the submissions of the Baseline students still outperform those of the Visual Reports Phase for the number of undetected mutants, the results suggest that the provided visualization helped close the gap between the Textual Reports Phase and the Baseline.

*5.4.2 Quality of Rationale on Equivalent and Productive Mutants.* In addition to writing tests, students were asked two conceptual questions: (1) to provide a proof for each mutant they believed to be equivalent and (2) to explain which (if any) mutants were unproductive. We qualitatively analyzed student responses to these questions, using the classification described in section 3.3.2.

We do not observe notable differences in performance across the two phases: the quality of comments for the Baseline is slightly better than the quality for the Textual Reports Phase, which is in turn slightly better than the quality for the Visual Reports Phase. While the visualizations may have made students more effective at mutation testing, we do not observe evidence that they improved conceptual understanding of equivalent and productive mutants. Further visual support specifically targeted at these concepts may improve the quality of student comments.

## 5.5 Complexity Challenge

As explored in Sec. 4, alleviating unnecessary complexity can help students focus and perform better. However, not all complexity is unnecessary. One of the goals of using the `Triangle` program for the Visualization Study was to provide a small program where each line of code can be fully reasoned about. This allows instructors to know ground truth information about the program such as the exact number of equivalent mutants and reduces the amount of code students need to understand. However, this simplification may hurt student learning. For instance, we observed students more focused on getting to the ground truth number of equivalent mutants than trying to really practice mutation testing, where in non-academic settings (and even in non-trivial academic settings) the ground truth number of equivalent mutants is not know [24, 32]. Additionally, the simplicity of the `Triangle` program makes it difficult for students to learn to reason about the productivity of mutants because with code that is so simple, reasoning about a mutant is less likely to advance code knowledge. Finding the correct balance of complexity and simplicity for educational mutation testing exercises will require further exploration, but exercise scaffolding and visual reports show promise in this direction.

## 6 Automatic Assessment Challenges

The test cases developed by the students in both the Scaffolding Study and the Visualization Study were executed to check their effectiveness and consistency. We observed that some submissions required manual changes to both the tests and infrastructures to be executed because they caused compile time or run time errors, thus affecting the execution and analysis of the tests. Tab. 3 summarizes the issues affecting the student test suites of both case studies; since some test suites had more than one issue, some submissions are reflected in multiple rows. Specifically, we observed the following issues that we manually corrected to automatically and consistently analyze all student submissions:

- **Failing test cases**: Student submissions that included tests which failed on unmutated code were corrected with the assumption that the unmutated is correct. Where fixes were non-trivial, tests were commented out. For the Scaffolding Study, D4JE was set to ignore test failures, so submissions that were assigned to Major are not included in this category.
- **Unexpected format/configuration**: Student submissions that did not follow the expected format or configuration were modified.
- **Compiler errors**: Student submissions that caused compiler errors were fixed to compile and run.
- **Missing assertions**: This issue was only relevant for the Visualization Study where an assignment question asked students to comment out a particular `assertEquals` and observe changes. Submissions that did not revert the commented out code were changed to do so.

These issues indicate areas that would cause friction with autograder use or otherwise having instructors run student test suites, even when students were provided with assignment infrastructure. These indicate areas for possible clarifications of instructions or additional improvements to infrastructure.

## 7 Limitations and Threats to Validity

Both studies presented in this work have limitations derived from being performed in common classroom settings. Many factors were not controlled between course editions, including but not limited to different instructors, general programming proficiency within and between different student populations, modifications to assignment instructions, and group work composition (e.g., students working in groups vs. individually for the Scaffolding Study). Thus, we refrain from making any strong claims about the generalizability or statistical significance of our observations, rather we report our observations and future directions for exploration.

## 8 Related Work

Mutation analysis and testing has been used in educational settings both as a way to support other learning goals (e.g., as an assessment measure of test suite quality) and as a primary learning goal (i.e. teaching modern testing strategies).

### 8.1 Using Mutation Analysis to Support Other Learning Goals

Mutation analysis has seen limited use in the context of student assessment and feedback [14]. Issues limiting its adoption include high computational costs [17]. These costs can delay feedback and thus degrade learning outcomes when used for automated assessment and feedback [3, 10]. In particular, [23] observed that these problems hindered research about the potential pedagogical benefits of mutation analysis. Despite these problems, several experiences of introducing mutation analysis for assessment and feedback of student test suite strength can be found in the literature.

Mutation analysis has seen adoption as an assessment measure of the quality of student tests. [38] used automatically generated mutants for this purpose while [39] used a set of buggy versions of the application under test. Mutation analysis provides a stronger measure of student test suite quality than code coverage measures alone [1]. In an empirical evaluation, [30] found a strong correlation between the mutation score from automatically generated mutants and manually-seeded faults. This experiment also found a moderately strong correlation between the mutation score and buggy student implementations, indicating that the mutation score from automatically generated mutants is a reasonable assessment measure of student test suite quality. Similarly, [6] also found that generated mutants are coupled to student-written program faults and that the mutation score is a reasonable measure of student test suite quality.

**Table 3: Number of issues with student test suites that disrupt automated assessment. Some submissions had multiple issues.**

| Issue | Number of Occurrences | | | | |
|---|---|---|---|---|---|
| | Scaffolding Study | | Visualization Study | | |
| | Direct | Scaffolding | Baseline | Textual | Visual |
| **Failing test cases** | 3 | 2 | 1 | 2 | 4 |
| **Unexpected format or configuration** | 6 | 6 | 0 | 3 | 0 |
| **Compiler errors** | 1 | 2 | 0 | 0 | 2 |
| **Missing assertions** | 0 | 0 | 2 | 4 | 4 |
| **Submissions that needed modification** | 9 / 33 (0.27) | 8 / 39 (0.21) | 3 / 24 (0.12) | 6 / 33 (0.18) | 9 / 43 (0.21) |

Mutation analysis has also seen adoption as a feedback measure of test suite quality for students. [9] ran an experiment for third year undergraduate computer science students in a Software Verification and Validation course. First, the students designed unit test cases without the guidance of any measurable testing criteria, such as statement or mutation coverage. Then, the students ran a mutation analysis tool on their test suite and observed the mutation score and used this feedback to design more effective test cases. Before and after running the mutation analysis, students were asked to assess how good the test suite was. After viewing the mutation analysis reports, student perception of the fault detection ability of their tests more closely aligned with the actual fault detection ability than before viewing the mutation analysis reports.

In comparing code coverage to mutation analysis as a quality metric for test suites, [27] found that students using mutation analysis wrote both stronger test suites and had higher quality code, as well as showed more evidence of following incremental testing practices.

Mutation analysis has also been used to support code comprehension. In [28, 29], first year undergraduate students used a mutation analysis tool to help reason about code; the goal was to understand the code functionality by trying various program inputs. When comparing students who used the mutation analysis tool with those who just had access to a compiler, students showed signs of having a better understanding of code functionality using the mutant tool.

Rather than using mutation analysis as a way to assess and provide feedback about student test suite quality or to support code comprehension, our work focuses on challenges and support with teaching mutation analysis and testing as a core learning goal.

### 8.2 Teaching Mutation Analysis and Testing as a Primary Learning Goal

Efforts to teach mutation testing as a key learning goal in software engineering and computer science courses has also been explored, including the development of a learning module [26].

[40] presented an educational mutation testing framework which uses a GUI on top of Defects4J to support use of different mutation testing tools through a common interface. We complement this work by presenting another mutation testing scaffolding tool and provide an evaluation of use of this tool in a classroom setting.

Code Defenders is a web-based game used for a gamification-based approach to teaching mutation testing [34, 35]. Code Defenders incorporates both test design and mutant generation by students. Some players act as "Attackers" and inject mutants in source code, while other players act as "Defenders" and try to design a test suite able to detect all the injected mutants. Empirical evaluations in a controlled study and a crowdsourcing scenario demonstrated stronger test suites and mutants created through game-play than those created by automated tools (i.e. Randoop and EvoSuite for test suites and Major for mutant generation) and that writing tests was considered more enjoyable in the game-based approach than outside this environment [36]. Code Defenders has additionally been used in educational settings [12, 13] with students enjoying the game and showing improvement in test suite and mutant quality throughout a semester [11].

Our work complements these explorations of teaching mutation analysis and testing as a primary learning goal in computer science and software engineering courses.

## 9 Conclusion

Through two case studies, we explore challenges to integrating mutation testing into software engineering curricula. We leave to future work further exploration on improving the application of mutant equivalency and productivity concepts. Additionally, we observe challenges to automatic assessment of student tests, even when assignment scaffolding is used, indicating another area to improve support for student and instructor use of mutation analysis tools and practicing mutation testing.

## 10 Data Availability

The study replication package is available at https://doi.org/10.6084/m9.figshare.28273163. This includes the experimental setup materials, anonymized consolidated datasets of analysis measures, and the analysis scripts used to generate the figures in this paper. We do not include the individual student submissions to preserve student privacy.

## Acknowledgments

This work is partially funded by ENACTEST (European iNnovation AllianCe for TESting educaTion), ERASMUS+ Project 101055874, 2022–2025 and by GATT (GAmification in Testing Teaching), funded by the University of Naples Federico II Research Funding Program (FRA). This work is supported in part by National Science Foundation grants CCF-1942055 and CNS-2120070.

## References

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, part of SPLASH 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 153–160. doi:10.1145/1869542.1869567

[2] Domenico Amalfitano, Ana C. R. Paiva, Alexis Inquel, Luís Pinto, Anna Rita Fasolino, and René Just. 2022. How do Java mutation tools differ? *Commun. ACM* 65, 12 (2022), 74–89. doi:10.1145/3526099

[3] Roger Azevedo and Robert M. Bernard. 1995. A Meta-Analysis of the Effects of Feedback in Computer-Based Instruction. *Journal of Educational Computing Research* 13, 2 (1995), 111–127. doi:10.2190/9LMD-3U28-3A0G-FTQT arXiv:https://doi.org/10.2190/9LMD-3U28-3A0G-FTQT

[4] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry - A Study at Facebook. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 268–277. doi:10.1109/ICSE-SEIP52600.2021.00036

[5] Peter J. Clarke, Andrew A. Allen, Tariq M. King, Edward L. Jones, and Prathiba Natesan. 2010. Using a web-based repository to integrate testing tools into programming courses. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, part of SPLASH 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 193–200. doi:10.1145/1869542.1869573

[6] Benjamin Simon Clegg, Phil McMinn, and Gordon Fraser. 2021. An Empirical Study to Determine if Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding. In *SIGCSE '21: The 52nd ACM Technical Symposium on Computer Science Education, Virtual Event, USA, March 13-20, 2021*, Mark Sherriff, Laurence D. Merkle, Pamela A. Cutter, Alvaro E. Monge, and Judithe Sheard (Eds.). ACM, 1055–1061. doi:10.1145/3408877.3432411

[7] Cobertura. 2022. Cobertura: A code coverage utility for Java. https://cobertura.github.io/cobertura/ Last accessed on 2025-01-22.

[8] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 449–452. doi:10.1145/2931037.2948707

[9] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Miguel Ángel Álvarez-García, and Kevin J. Valle-Gómez. 2021. Mutation Testing and Self/Peer Assessment: Analyzing their Effect on Students in a Software Testing Course. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 231–240. doi:10.1109/ICSE-SEET52601.2021.00033

[10] Fabienne M. Van der Kleij, Remco C. W. Feskens, and Theo J. H. M. Eggen. 2015. Effects of Feedback in a Computer-Based Learning Environment on Students' Learning Outcomes: A Meta-Analysis. *Review of Educational Research* 85, 4 (2015), 475–511. doi:10.3102/0034654314564881 arXiv:https://doi.org/10.3102/0034654314564881

[11] Gordon Fraser, Alessio Gambi, Marvin Kreis, and José Miguel Rojas. 2019. Gamifying a Software Testing Course with Code Defenders. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, Elizabeth K. Hawthorne, Manuel A. Pérez-Quiñones, Sarah Heckman, and Jian Zhang (Eds.). ACM, 571–577. doi:10.1145/3287324.3287471

[12] Gordon Fraser, Alessio Gambi, and José Miguel Rojas. 2018. A Preliminary Report on Gamifying a Software Testing Course with the Code Defenders Testing Game. In *Proceedings of the 3rd European Conference of Software Engineering Education, ECSEE 2018, Seeon Monastery, Bavaria, Germany, June 14-15, 2018*, Jürgen Mottok (Ed.). ACM, 50–54. doi:10.1145/3209087.3209103

[13] Gordon Fraser, Alessio Gambi, and José Miguel Rojas. 2020. Teaching Software Testing with the Code Defenders Testing Game: Experiences and Improvements. In *13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 461–464. doi:10.1109/ICSTW50294.2020.00082

[14] Vahid Garousi, Austen Rainer, Per Lauvås Jr., and Andrea Arcuri. 2020. Software-testing education: A systematic literature mapping. *J. Syst. Softw.* 165 (2020), 110570. doi:10.1016/J.JSS.2020.110570

[15] Gregory Gay and Alireza Salahirad. 2023. How Closely are Common Mutation Operators Coupled to Real Faults?. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*. IEEE, 129–140. doi:10.1109/ICST57152.2023.00021

[16] Sean A. Irvine, Tin Pavlinic, Leonard Trigg, John G. Cleary, Stuart Inglis, and Mark Utting. 2007. Jumble Java Byte Code to Measure the Effectiveness of Unit Tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 169–175. doi:10.1109/TAIC.PART.2007.38

[17] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[18] René Just. [n. d.]. The Major Mutation Framework. https://mutation-testing.org. Last accessed on 2023-11-16.

[19] René Just. 2014. The major mutation framework: efficient and scalable mutation analysis for Java. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 433–436. doi:10.1145/2610384.2628053

[20] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. doi:10.1145/2610384.2628055

[21] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 612–615. doi:10.1109/ASE.2011.6100138

[22] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing Mutants to Guide Mutation Testing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1743–1754. doi:10.1145/3510003.3510187

[23] Ayaan M. Kazerouni, James C. Davis, Arinjoy Basak, Clifford A. Shaffer, Francisco Servant, and Stephen H. Edwards. 2021. Fast and accurate incremental feedback for students' software tests using selective mutation analysis. *J. Syst. Softw.* 175 (2021), 110905. doi:10.1016/J.JSS.2021.110905

[24] Benjamin Kushigian, Samuel J. Kaufman, Ryan Featherman, Hannah Potter, Ardi Madadi, and René Just. 2024. Equivalent Mutants in the Wild: Identifying and Efficiently Suppressing Equivalent Mutants for Java Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 654–665. doi:10.1145/3650212.3680310

[25] Lech Madeyski and N. Radyk. 2010. Judy - a mutation testing tool for Java. *IET Softw.* 4, 1 (2010), 32–42. doi:10.1049/IET-SEN.2008.0038

[26] J.C. Maldonado and E.F. Barbosa. 2006. Establishing a Mutation Testing Educational Module based on IMA-CID. In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*. 14–14. doi:10.1109/MUTATION.2006.4

[27] Rifat Sabbir Mansur, Clifford A. Shaffer, and Stephen H. Edwards. 2024. Mutating Matters: Analyzing the Influence of Mutation Testing in Programming Courses. In *Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, SIGCSE Virtual 2024, Virtual Event, NC, USA, December 5-8, 2024*, Mohsen Dorodchi, Ming Zhang, and Stephen Cooper (Eds.). ACM. doi:10.1145/3649165.3690110

[28] Rafael A. P. Oliveira, Lucas B. R. Oliveira, Bruno B. P. Cafeo, and Vinicius H. S. Durelli. 2014. On using mutation testing for teaching programming to novice programmers. In *International Conference on Computers in Education*. doi:10.58459/icce.2014.413

[29] Rafael A. P. Oliveira, Lucas B. R. Oliveira, Bruno Barbieri Pontes Cafeo, and Vinicius H. S. Durelli. 2015. Evaluation and assessment of effects on exploring mutation testing in programming courses. In *2015 IEEE Frontiers in Education Conference, FIE 2015, El Paso, TX, USA, October 21-24, 2015*. IEEE Computer Society, 1–9. doi:10.1109/FIE.2015.7344051

[30] James Perretta, Andrew DeOrio, Arjun Guha, and Jonathan Bell. 2022. On the use of mutation analysis for evaluating student test suite quality. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 263–275. doi:10.1145/3533767.3534217

[31] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 910–921. doi:10.1109/ICSE43902.2021.00087

[32] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Trans. Software Eng.* 48, 10 (2022), 3900–3912. doi:10.1109/TSE.2021.3107634

[33] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 47–53. doi:10.1109/ICSTW.2018.00027

[34] José Miguel Rojas and Gordon Fraser. 2016. Code Defenders: A Mutation Testing Game. In *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 162–167. doi:10.1109/ICSTW.2016.43

[35] José Miguel Rojas and Gordon Fraser. 2016. Teaching mutation testing using gamification. In *European Conference on Software Engineering Education (ECSEE)*.

http://www.code-defenders.org/papers/ECSEE16_MutationEducation.pdf

[36] José Miguel Rojas, Thomas D. White, Benjamin S. Clegg, and Gordon Fraser. 2017. Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 677–688. doi:10.1109/ICSE.2017.68

[37] Ana Belén Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. 2022. Mutation testing in the wild: findings from GitHub. *Empir. Softw. Eng.* 27, 6 (2022), 132. doi:10.1007/S10664-022-10177-8

[38] Zalia Shams and Stephen H. Edwards. 2013. Toward practical mutation analysis for evaluating the quality of student-written software tests. In *International Computing Education Research Conference, ICER '13, La Jolla, CA, USA, August 12-14, 2013*, Beth Simon, Alison Clear, and Quintin I. Cutts (Eds.). ACM, 53–58. doi:10.1145/2493394.2493402

[39] Rebecca Smith, Terry Tang, Joe Warren, and Scott Rixner. 2017. An Automated System for Interactively Learning Software Testing. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2017, Bologna, Italy, July 3-5, 2017*, Renzo Davoli, Michael Goldweber, Guido Rößling, and Irene Polycarpou (Eds.). ACM, 98–103. doi:10.1145/3059009.3059022

[40] Pedro Tavares, Ana C. R. Paiva, Domenico Amalfitano, and René Just. 2024. FRAFOL: FRAmework FOr Learning mutation testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1846–1850. doi:10.1145/3650212.3685306

[41] Dávid Tengeri, László Vidács, Árpád Beszédes, Judit Jász, Gergö Balogh, Béla Vancsics, and Tibor Gyimóthy. 2016. Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density. In *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 174–179. doi:10.1109/ICSTW.2016.25

[42] Victor Veloso and André C. Hora. 2022. Characterizing High-Quality Test Methods: A First Empirical Study. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 265–269. doi:10.1145/3524842.3529092