**MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING**

# MuTEd

## A Comparative Study of Classic and Extreme Mutation Testing for Teaching Software Testing

LUYCX, Pierre; Balfroid, Martin

*Award date:*
2022

*Awarding institution:*
University of Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

# MuTEd: A Comparative Study of Classic and Extreme Mutation Testing for Teaching Software Testing

Balfroid Martin

Luycx Pierre

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR (BELGIUM)

**Résumé**

Bien que le test soit une activité essentielle du génie logiciel, des études ont montré un écart important entre les connaissances des étudiants et les besoins de l'industrie en la matière. Cela souligne la nécessité d'explorer de nouvelles approches pour enseigner le test logiciel. Parmi celles-ci, le *mutation testing* classique s'est déjà avéré efficace pour aider les étudiants. Nous supposons que le *mutation testing* extrême pourrait être plus efficace, car il génère des mutants qui seraient plus évidents à tuer. Afin d'étudier cette question, nous avons organisé une expérience avec deux classes de premier cycle comparant l'utilisation de deux outils, l'un appliquant du *mutation testing* classique et l'autre du *mutation testing* extrême. Les résultats ont contredit notre hypothèse. En effet, les étudiants ayant accès à l'outil de *mutation testing* classique ont obtenu un meilleur score de mutation, tandis que les autres semblent surtout avoir couvert plus de code. Enfin, nous avons publié et annonymisé les suites de tests des étudiants dans le respect des bonnes pratiques de *l'open-science*, et nous avons élaboré des conseils sur base de nos résultats et des travaux antérieurs.

**Mots-clés :** *Software Testing, Software Testing Education, Mutation Testing, Extreme Mutation Testing*

**Abstract**

Although software testing is critical in software engineering, studies have shown a significant gap between students' knowledge of software testing and the industry's needs, hinting at the need to explore novel approaches to teach software testing. Among them, classical mutation testing has already proven to be effective in helping students. We hypothesise that extreme mutation testing could be more effective by introducing more obvious mutants to kill. In order to study this question, we organised an experiment with two undergraduate classes comparing the usage of two tools, one applying classical mutation testing, and the other one applying extreme mutation testing. The results contradicted our hypothesis. Indeed, students with access to the classic mutation testing tool obtained a better mutation score, while the others seem to have mostly covered more code. Finally, we have published and anonymised the students' test suites in adherence to best open-science practices, and we have developed guidance based on previous evaluations and our own results.

**Keywords:** *Software Testing, Software Testing Education, Mutation Testing, Extreme Mutation Testing*

**Acknowledgements**

We would like to thank all the people who have contributed directly or indirectly to the realisation of this master thesis:

First, we show our gratitude to Julie Henry who reviewed our experimental design and helped improve it, to Julien Albert and Thibaut Septon who gave very valuable feedback on our work and allowed us to see the mistakes we did not see, and to Patrick Heymans and Tony Leclercq who very kindly allowed us to take some of their lecture time to carry out our experiment.

Then, we would also like to thank Antoine, Basile, and Maxime who helped us calibrating the experience and let us foresee what we had not foreseen. More generally, we would like to thank all the anonymous students who participated in the experiment and without whom nothing would have been possible.

We want to express our thanks to our relatives who supported us during the research, design, execution, and writing of this work.

But, most of all, we would like to thank our research supervisors, Xavier Devroey and Benoît Vanderose, for their regular help, their helpful proofreading, their quick and wise answers, and, finally, their welcome to the team during our internship.

# Contents

# Chapter 1

# Context

## 1.1 Introduction

Today, activities rely on software, whether it is economics, health, commerce, education, energy, banking, marketing, design, entertainment, transportation, management, communication, research, etc. If some applications are without real consequences, others are critical. Society, money, and even lives all depend on software. However, we cannot assume that all software is reliable. Verifying if a piece of code is trustworthy is the goal of software testing and is an essential skill for computer science students to learn [1].

Yet, studies have shown gaps, or rather mismatches, in computer science students' knowledge relative to industry needs. In particular, software testing is one of the main subjects of software engineering where such deficiencies are to be found. Practical activities seem to be missing more than theoretical ones, mainly in web application testing, functionality testing, and deriving tests from client's requirements [2].

Therefore, original methods of software testing education must be explored. Among others, some novel approaches in both software engineering education and software testing include using *mutation testing*, the *RIP model*, and *gamification*.

In this master thesis, we explore the usage of *Reneri* [3] as a means to teach unit testing to undergraduate students. *Reneri* is an *extreme mutation testing* tool that uses automatic fault infection and propagation analysis. It can automatically generate text reports that help improve an existing test suite.

*Classical mutation testing* introduces fine-grained modifications in the source code, assuming it represents minor faults a real programmer could do. In contrast, *extreme mutation testing* creates coarse-grained modifications, removing entire method bodies at once. Thus, we suppose that extreme mutation testing could generate "more obvious" mutants to kill for learning students.

We thus ran a comparative experiment between *Reneri* and *PIT* (a common mutation testing tool for Java) with learning students. In the following chapters, we will present previous works, detail the experiment process, provide the results, and discuss them.

## 1.2 Contributions

The contributions of this master thesis are the following:

1. A comparison between classic mutation operators and extreme mutation operators in the context of software testing education.
2. Recommendations on how to use a mutation testing tool in a testing course based on our results and previous evaluations.
3. Lessons learned from conducting such an experiment with students.
4. Open Data. Our results contain Java test code written by students, their mutation score, instruction coverage, automatically generated reports, as well as answers to a questionnaire and self-evaluations made by the students. These are all anonymised. See Chapter 4 for more information.

# Chapter 2

# Background

## 2.1 Why Do We Test Software?

"To err is human", as the famous Latin expression goes. Yet, programmers are also human beings and are thus prone to error. However, Cicero would add that "only the fool persists in his fault". So, programmers being (generally) reasonable people, it is not surprising that they have been trying to find ways for years to know if they did not make any of these errors which can sometimes be catastrophic. Determining whether a program behaves as expected is called *software testing*, which should be distinguished from *debugging*, i.e., determining the cause of the error and removing it [4].

Naively, beginners will often resort to print statements to check if their program is working correctly. For instance, in Listing 2.1, to check that *list.size()* is returning 2, a novice programmer would write a print statement like in line 6 and look at the console to see if "2" is actually displayed.

Listing 2.1: Manual Testing of *list.size()*

```
1 public class Main {
2     public void main(String[] args) {
3         MyList list = new MyList();
4         list.add(5);
5         list.add(23);
6         System.out.println(list.size()); // Should print "2"
7     }
8 }
```

However, this is a poor practice because, for a more complex program with thousands of lines of code, this process quickly becomes tedious, if not impossible. Moreover, it must be repeated after each modification of the source code. Thus, experienced programmers rely on tools to automate this process instead. These

tools use *assertions* to check if programs behave as expected, failing in case of error. This is called *test automation* [4]. Listing 2.2 shows a more suitable version of the same example test.

Listing 2.2: A JUnit Test Case for *list.size()*

```
1     @Test
2     public void testMyListSize() {
3         MyList list = new MyList();
4         list.add(5);
5         list.add(23);
6         assertEquals(2, list.size());
7     }
```

## 2.2 What Types of Testing Techniques Exist?

In his book "Foundations of Software Testing", A. Mathur [4] proposed to classify testing techniques according to a set of five classifiers:

1. The source of the test generation. This is usually the requirements or the source code. In the first case, we talk about *black-box* testing and *white-box* testing in the second.

2. The phase of the software life cycle during which testing takes place. During coding, we talk about *unit* testing. It is called *integration* testing during the integration phase, *system* testing during the system integration phase, *regression* testing during maintenance, and *beta-testing* after the release. Unit testing, which will be the concern of this work, focuses on small independent units, such as functions or methods in a source code.

3. The goal of the specific testing activity. They can be grouped into two broad categories: *functional* and *non-functional* testing. The first group targets the software's functionalities. The second concerns safety, reliability, performance, localisation, acceptability, etc.

4. The characteristics of the artefact under test. For instance, we use the term *component* testing if we focus on application components, *client-server* testing for a client-server architecture, or *object oriented* testing for an object oriented software.

5. The test process model. Testing can be integrated into different development processes. In the *waterfall* model, testing is done last. In the *V-model*, testing is done at each phase. In *spiral testing*, it is applied to each increment.

*Agile* testing (in agile methodologies) give specific guidelines. And finally, *test driven development* (TDD) requires writing tests before the code, as they represent the requirements.

In this master thesis, we will only focus on *white-box functional unit* testing. Regarding the characteristics of the artefact under test which will be described in Section 3.1.3, we will be looking at *object oriented* testing. Finally, our study does not fit into a specific development process model.

## 2.3 How to Write a Good Test?

In the book "Clean Code: A Handbook of Agile Software Craftsmanship", R.C. Martin [5] devotes a chapter to writing "clean" unit tests. He insists that test code is just as critical as production code because it is what makes it flexible, maintainable, and reusable. It reduces the fear of inadvertently introducing a bug when altering the code. He also emphasizes readability, which he says is more important in a unit test than in production code. What makes a test readable is the same as any code: clarity, simplicity, and the bare minimum of expressions. He presents five rules for writing good tests, known by the easy-to-remember acronym FIRST:

1. A test must be **Fast** because if it is not, developers are less likely to run it frequently, and thus to find problems quickly.

2. A test must be **Isolated** because it must deal with only one feature and should not be dependent on other tests. Indeed, if a test depends on another one, it will crash if the previous test crashes, making the diagnosis more difficult.

3. A test must be **Repeatable**, i.e., it must give the same result every time to build confidence in the testing process. For instance, randomness should be avoided in a test.

4. A test must be **Self-validating**, i.e., it must assert that both the output values and the final state are correct.

5. A test must be written **Timely**, i.e., the test should be written before or shortly after production. Waiting any longer almost guarantees that it will never be written.

R. Osherove [6] proposes quite similar properties of a good unit test. He adds that it must be easy to implement, easy to identify the problem in case of failure, and that the test must have total control of the unit under test.

One way to improve the readability of a test is to standardize the way a test is structured. The most common structure is by splitting the unit test into three parts: (1) setting up the initial state, (2) operate on the initial state, and (3) check that the resulting state is the one expected. In the literature, there are various names for these three steps such as *Arrange, Act, Assert* (AAA) [7], *Given, When, Then* [5], or *Build, Operate, Check* [5].

Listing 2.3: A Example Test Case Structured According to AAA

```
1   @Test
2   public void testMyListSize() {
3       // Arrange
4       MyList list = new MyList();
5       list.add(1);
6
7       // Act
8       int size = list.size();
9
10      // Assert
11      assertEquals(1, size);
12  }
```

## 2.4 Who Will Test the Tests?

When one runs a program against a test suite and all the tests pass, it does not mean the program is correct. Indeed, if a test fails, it proves the presence of a defect. However, if no tests fail, it does not prove the absence of all defects [1]. To even consider such a conclusion, one would have to write a practically infinite number of tests. Therefore, an important question in software testing research is how to evaluate whether a test suite is complete and adequate [1].

It is impossible to have a completely exhaustive test suite, as this would require an enormous number of tests. However, regarding the adequacy of a test suite, scientists have proposed many criteria to measure it quantitatively over the years. Indeed, even if the total input domain is vast, and effectively infinite, it can be subdivided into smaller input domains. Each input in one input domain is considered to be equivalent, and each of these input domains represents a *test requirement* that must be covered by the test suite. This makes testing feasible [1].

Test requirements can take several forms in source code. They are criteria that are generally used as adequacy metrics, objectives that testers should attain to achieve a certain level of trust in a program.

### 2.4.1 Control Flow Adequacy Criteria

The most common criteria for evaluating the adequacy of a test suite are based on control flow coverage. One of its simplest form is *line coverage*. The idea is to calculate the ratio of executed lines to the total number of lines in the source code when the test suite is run. There also exist other simple forms of coverage: *statement coverage*, *block coverage*, *method coverage* focuses respectively on statements, blocks, and methods in the source code. More elaborated forms also exist. We can think of *branch coverage* where we consider the two outcomes for each if-statement. There are also forms of *path coverage* where every possible path should be covered, with the limitation that there can exist unfeasible paths [1]. To overcome this limitation, *basic path coverage* [8] can be used. It considers the coverage of the basic paths, i.e., paths which, taken in combination, generate all possible paths. It is related to *cyclometric complexity* [9], a measure of complexity.

These techniques rely on code instrumentation. During compilation, special instructions are inserted at certain locations. When executed, they record that some sort of node (method, block, line, statement, branch, etc.) is covered. The test suite is executed against this instrumented version of the program, and results can then be reported.

It has been shown that some forms of coverage are stronger than others in terms of revealing faults [10]. For example, branch coverage yields higher fault revelation than statement coverage. The same study shows that "mutation testing" (Section 2.4.2 below) is more effective that control flow criteria.

### 2.4.2 Mutation Testing

Mutation testing is a testing technique for assessing the quality of a test suite by artificially introducing defects, called *mutants*, into the source code of a program [11]. The test suite is then run against these mutants. On the one hand, if at least one test fails, the mutant is said to be *killed* or *detected*. On the other hand, if no test fails, the mutant is said to be *live*. In this case, the test suite must be improved to kill the mutant.

Defects are inserted using *mutation operators* that generate small variations in the source code. A typical example is to switch a "+" into a "−" in an arithmetic expression. A basic set proposed by Offutt et al. [12] can be seen in Table 2.1 with some examples applied to Java code. The original code is presented on top and its mutant below. This set is usually considered a minimum standard for mutation testing [11].

After running a test suite against the mutants, its *mutation score* can be calcu-

Table 2.1: Basic Operators Set with Non-Exhaustive Examples

| Name | Example Mutant |
|---|---|
| Absolute Value Insertion (ABS) | `return x;`<br>`return Math.abs(x);` |
| Arithmetic Operator Replacement (AOR) | `int b = x + y;`<br>`int b = y;` |
| Logical Connection Replacement (LCR) | `if (x >= 0 && x < 10)`<br>`if (x >= 0 \|\| x < 10)` |
| Relational Operator Replacement (ROR) | `for (int i = 0; i < size; i++)`<br>`for (int i = 0; i <= size; i++)` |
| Unary Operator Insertion (UOI) | `return x * x;`<br>`return x * -x;` |

lated. It is the percentage of mutants killed out of the total number of mutants. It can be used to measure test completeness and can also be used as an adequacy metric [1]. Roughly speaking, mutation score tells (1) what to test (2) when to stop testing and (3) if a test suite can be trusted. Chekam et al. [10] empirically showed that mutation testing leads to high fault revelation whereas statement and branch testing do not.

Unfortunately, not all mutants have the same value, which biases the final mutation score [13]. On the one hand, some mutants, called *equivalent mutants*, are impossible to kill as they return the same output values and have the same side effects as the original program [1]. Detecting these mutants is one of the main challenges in mutation testing [13, 14], as it is an undecidable problem [15]. Listings 2.4 and 2.5 shows an example of an equivalent mutant.

On the other hand, some mutants do not contribute to the testing process because the test suite kills them along with other mutants. These types of mutants are said to be *redundant*. Papadakis et al. [14] identify two subtypes of *redundant mutants*: *duplicated mutants* and *subsumed/joint mutants*. *Duplicated mutants* are equivalent to each other but not to the original program [16]. *Subsumed* or *joint mutants* are mutants who, when another mutant is killed, are also killed [13, 17].

Mutation testing is effective in revealing a software's faults. This effectiveness comes from 3 different assumptions:

- Under the *Competent Programmer Hypothesis* [18], programmers create nearly correct programs. Only a few minor syntactic changes are necessary to get the correct version. Consequently, if a test suite can detect all generated defects, it proves that such defects are not present in the system under test (SUT). The creation of minor variations simulates the type of frequent defects

Listing 2.4: Original Code

```
1 public class Scoreboard {
2     private int highscore = 0;
3
4     public void updateScore(int newScore) {
5         if (newScore > this.highscore) {
6             this.highscore = newScore;
7         }
8     }
9 }
```

Listing 2.5: An Equivalent Mutant

```
1 public class Scoreboard {
2     private int highscore = 0;
3
4     public void updateScore(int newScore) {
5         if (newScore >= this.highscore) {
6             this.highscore = newScore;
7         }
8     }
9 }
```

we seek.

- Due to the *Mutant Coupling Effect* [19], a test suite that reveals simple defects may reveal more complex ones. Indeed, the coupling between first-order mutants (i.e., mutants with only one variation) and complex mutants (multiple variations) is such that killing simple mutants reveals a large proportion of complex mutants. This phenomenon has been studied theoretically and practically.

- Based on the *RIPR model* [20] (detailed in Section 2.4.4), designing test cases specifically to kill mutants encourages developers to place assertions at each point where a failure could occur. This makes the test cases more robust because, in automated testing, failures can only be revealed by assertions.

### 2.4.3 PIT

*PIT* [21], alternatively known as *PITest*, is an open-source[1] mutation testing tool for Java. It operates directly on Java virtual machine (JVM) bytecode and it is well

[1]It is available at `https://pitest.org/`.

integrated with development tool such as *Maven*, making it fast and applicable to real-world software. Furthermore, it is actively developed and maintained, not like other tools.[2] PIT uses a *mutation engine* to generate mutants based on mutation operators. Its default mutation engine is called *Gregor*.

### 2.4.4 Reachability, Infection, Propagation, and Revealability

According to the *Reachability, Infection, Propagation model* (or *RIP model*) [22], there are three necessary conditions for a program fault to be detected. Firstly, the fault should be *reached* during the execution. Secondly, it should *infect* the program by changing its state. Lastly, this state change should *propagate* to the program's output.

Li et al. [20] proposed an extended version of the RIP model called *RIPR*, where the additional "R" stands for *Revealability*. As the name implies, the first three conditions stay the same. The added condition states that a fault must not only be propagated but also *revealed*. When talking about automated testing (and more specifically unit testing), revealing a fault implies writing an assertion that covers it. Indeed, a fault could propagate to the program state of a test case, but if no assertion reveals it, the test misses the point.

The standard condition for killing a mutant is that the mutant has an observable difference in the output of the program. In that case, we say that the mutant is *strongly* killed. However, this condition could be relaxed by asserting that only the program state has changed without necessarily propagating the changed state to the output. If the program state comparison is performed immediately after a mutant is executed, we say that the mutant is *weakly* killed. Otherwise, if it is performed at a later point, we say that the mutant is *firmly* killed. One might expect that weak and firm mutations would be less effective than strong mutations and would require fewer tests to satisfy coverage. Yet, experimentation has shown that, in reality, the difference is minimal in most cases. From the RIP model's perspective, killing a mutant satisfies all three conditions strongly, while killing a mutant weakly or firmly satisfies only the first two: reachability and infection [14, 1].

### 2.4.5 Pseudo-Tested Methods and Extreme Mutation

Niedermayr et al. [23] introduced pseudo-tested methods and extreme mutation operators as a way to detect them. A method is pseudo-tested when the test suite

[2] See `https://pitest.org/java_mutation_testing_systems/`.

covers it, yet no test fails when removing its body. This means that the test suite is only testing this method side effects superficially.

Extreme mutation operators are the operators that completely remove void methods' bodies and replace non-void methods with trivial return statements. A basic example can be seen in Listings 2.6 and 2.7. These operators have been created to automate the detection of pseudo-tested methods.

The authors applied extreme mutation to 14 Java open-source projects in their study. They found that, on average, 11.41% of methods are pseudo-tested in unit tests and 35.48% in system tests.

Vera-Pérez et al. introduced *Descartes* [24], an alternative mutation engine for PIT replacing Gregor, its original mutation engine (see Section 2.4.3). Descartes specializes in creating extreme mutants for the Java programming language. Using Descartes, the same authors reproduced Niedermayr's experiment on 21 subjects [25]. They drew similar conclusions and found that 1% up to 46% of methods are pseudo-tested in their dataset. Furthermore, they directly asked the developers for feedback and confirmed that some of these methods are troublesome.

The issue with pseudo-tests is that, while being covered, the methods are not well tested and could cause failures. They are executed, yet the test cases do not assess their effects. It also means that code coverage at the method level is not a valid metric to measure a test suite's quality.

With the mutant in Listing 2.7 above, one way this method could be pseudo-tested is when no test actually calls *equals* with two different instances of the class. In that case, every call to *equals* in every test case will always return true. Thus, replacing the entire method body with "`return true;`" does not change the outcome of the test suite's execution.

According to the RIPR model (Section 2.4.4), we can say that the fault is *reached* but does not cause an *infection* to the program state. To fix that, we must add a new test where *equals* returns false, *propagate* this return value to the test case, and finally *reveal* it with an assertion.

### 2.4.6 Reneri

Vera-Pérez et al. [3] created Reneri, a tool that leverages Descartes to generate automatic reports based on test execution and the RIPR model. It uses static analysis, bytecode manipulation, source code modification, and code instrumentation. These reports provide insightful advice for improving a test suite. The tool follows a similar reasoning to what we just did in the previous section.

First, Descartes is used to generate and run extreme mutants. This step produces the list of all surviving mutants. Then, Reneri instruments methods by

Listing 2.6: Original Code

```
1  public class Amount {
2      private int value = 0;
3
4      @Override
5      public boolean equals(Object that) {
6          if (that == null) return false;
7          if (!(that instanceof Amount)) return false;
8          return this.value == ((Amount)that).value;
9      }
10 }
```

Listing 2.7: Same Code with an Extreme Mutation Applied

```
1  public class Amount {
2      private int value = 0;
3
4      @Override
5      public boolean equals(Object that) {
6          return true;
7      }
8  }
```

modifying their bytecode using *Javassist* [26] and tests by modifying their source code using *Spoon* [27]. Instrumentation is used to observe the program's state: during execution, at the very end of each method, the state is observed and dumped into a file. More specifically, the observed state consists of the instance on which a method is invoked (if not static), every argument of the method, and its return value. Tests are also instrumented to observe the program state at every possible location.

Then, Reneri generates hints that are related to the RIPR model (see Section 2.4.4). Non-mutated and mutated methods are run against the test suite. For each mutated method, the program state is observed and compared to the program state without mutants. If there is no observable difference in the program state whether we use a mutant or not, then the *infection* condition is not fulfilled. If there is an observable difference in the method but not in the tests, it means that the fault is not *propagated*. Moreover, if there is an observable difference in a test, but no assertion fails, then the fault is not *revealed*. Therefore, there exist 3 different kinds of hints: *not-infected* hints, *not-propagated* hints, and *not-revealed* hints. Please note that the tool does not generate any hint for the *reached* condition.

Finally, these hints are used to produce a human-readable report. For each

hint, it contains the problem, a diagnosis, and a suggested solution. An example of such a report can be seen in Figure 2.1. In this case, it actually is a *not-infected* hint.

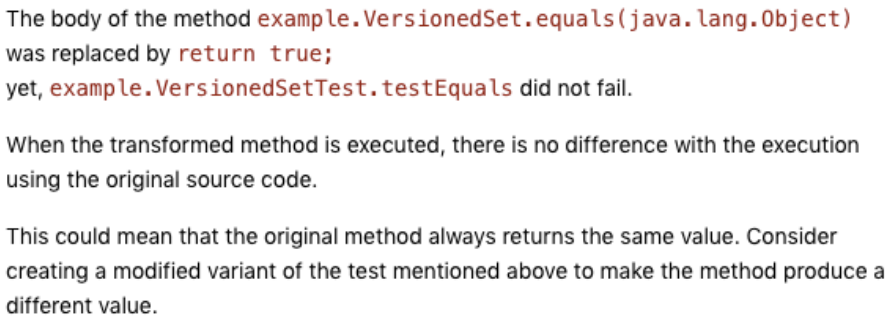



Figure 2.1: Example Report from Reneri [3]

Reneri has been evaluated with open-source software [3]. Fifteen projects were used as study subjects. For four of them, the developers were contacted and presented with some generated suggestions about their respective projects. It has been shown that the suggestions are helpful and could even point to the exact solution in some cases.

## 2.5 How to Teach Software Testing?

Scatalon et al. [2] made a survey aimed at Brazilian practitioners. The authors found that there were gaps between what is learned and what is actually used in industry concerning software testing. The study separated theoretical and practical knowledge and noted that most subjects are facing negative gaps on the theoretical side, and that all subjects are facing larger negative gaps on the practical side. A negative gap means that a topic is not presented in sufficient detail in relation to what is expected in the industry. On the contrary, some theoretical subjects, such as finite state machines and control flow graphs, show positive gaps, which means that the respondents feel that there is too much focus on these. We thus think that there is a need for innovative ways to teach software testing.

### 2.5.1 Common Challenges

Garousi et al. [28] conducted a systematic literature mapping (SLM) of over 200 articles between 1992 and 2019 about software testing in education. This review aims to help educators identify best practices for their software testing courses. It also aims to help researchers further research the subject.

From the reviewed papers, the authors gathered common challenges and insights on how to meet them. Some challenges come from the students' point of view, others from that of the instructors, and some come from both. The paper modeled these and their relationships in Figure 2.2.

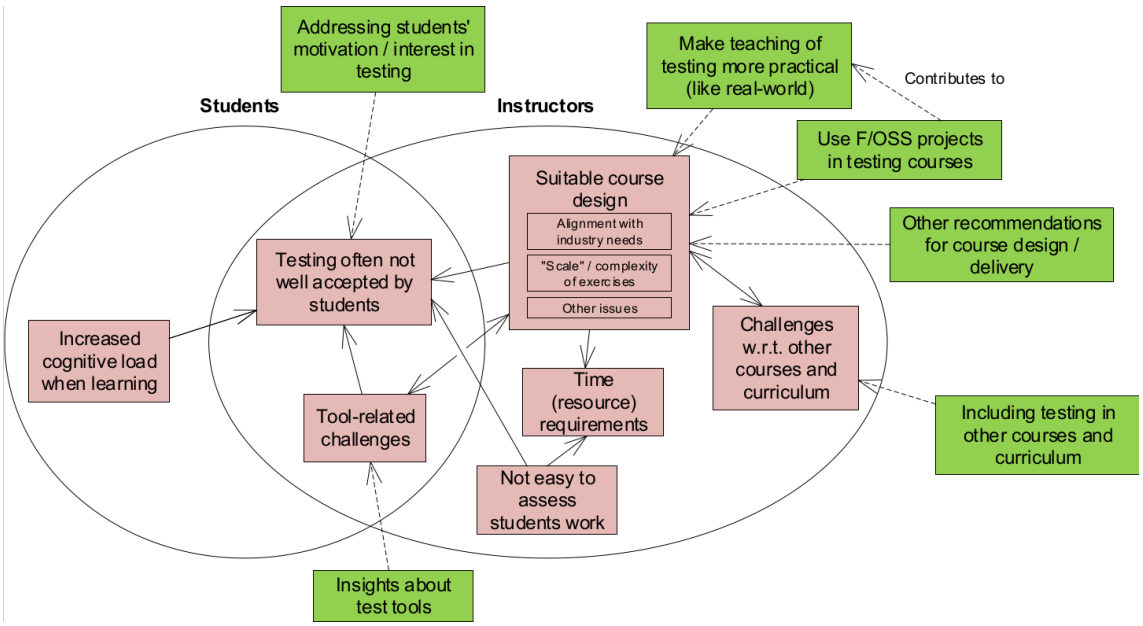



Figure 2.2: Challenges and Insights About Software Testing Education [28]

Generally, it has been observed that software testing is not well accepted by students. Indeed, they often feel that writing unit tests is tedious or even boring. In addition, simultaneously learning to program and to test represents a significant cognitive load for students. Another challenge for novice testers is the use of new tools. Lastly, students often think that testing a small software is uninteresting while testing a large-scale software is intimidating.

A key challenge for instructors is integrating software testing into other courses and allocating (or even finding) the time required. Secondly, assessing students' tests is complex. Automated grading has been used, but coverage testing is not a suitable metric. Mutation testing could help in that regard, but there is still a lack of adequate tools.

The reviewed articles can give us insights into how to meet these challenges. Gamification has been used to address students' motivation. Such examples include, among others, *CodeDefenders* [29] where some students introduce faults (playing as *attackers*) and the remaining players write unit tests (playing as *defenders*). The goal is to create as many surviving mutants for the first team and, for the latter, to achieve the highest mutation score. The authors found that students were actively involved in the game and that they improved their testing skills.

Another example is the *JPacman* framework [30] used in a testing course at TU Delft in the Netherlands. The idea is to give students a gamified SUT, which contains bugs on purpose. The students are assessed based on the quality of the test suite they write for this Pacman implementation. This paper's goal is to highlight common mistakes and challenges, hard topics to learn, and favourite learning activities. Thus, *JPacman* has not been directly evaluated. Yet, it is open-source[3] and we think it is an inspiring way to bring gamification to software testing education through the SUT.

### 2.5.2 Usage of Mutation Testing

First, some studies [31, 32, 33] showed that mutation testing can help towards better test suites for safety-critical software. The study cases were an aeronautical software system, a mechatronic software system, and the Linux kernel. In more general terms, studies tend to show a correlation between high mutation scores and fault detection rates [11]. Therefore, mutation testing helps with writing better test suites.

More specifically, in the field of education, Oliveira et al. [34] conducted an experiment with novice students. The authors compared two groups of students. The control group used a classical Pascal compiler, while the treatment group used "Pascal Mutants". In brief, this tool allows to create and manage test cases for the SUT, and generate and run mutants. All students were given a source code with no documentation and could run it with any inputs. Afterwards, they were asked to answer a questionnaire to see if they understood the algorithm. Students using mutation testing gave more correct answers than the other group. The authors also surveyed senior students about their opinion on mutation testing for teaching programming fundamentals. They concluded that mutation testing is beneficial in teaching novice students the basics of programming.

By combining mutation testing, self-assessment, and peer-assessment in their software testing course, Delgado-Pérez et al. [35] were able to get students to perceive the benefits of writing quality tests. In this experiment, students were given a C++ SUT and participated in two successive testing sessions. In the first session, they manually wrote unit tests without being guided by any coverage criteria. In the second session, they improved their existing test suite with access to the mutation score. After each session, students were asked to carry out two evaluations: one on the test suite they had written (self-assessment), and the other on the test suite another student had written (peer-assessment). Finally, students completed a questionnaire adapted from Oliveira et al. [34]. The results show

[3] And available at `https://github.com/serg-delft/jpacman-framework`.

that exposure to the mutation score made students aware of the need of using advanced testing techniques.

Finally, we already presented *CodeDefenders* [29] in the previous section (Section 2.5.1). It can be seen as a gamification of mutation testing where players manually create and kill mutants in teams.

Extreme mutation testing was not yet used in the context of software testing education. We think however that it may be an effective way to teach testing. Indeed, extreme mutation operators generate coarse-grained mutants by removing methods' entire bodies, in contrast to classical mutation testing which generates much more mutants which are fine-grained. Consequently, we expect that extreme mutants would be more obvious, more easy, and more accessible to novice computer science students.

# Chapter 3

# Experimental Evaluation

## 3.1 Experimental Design

So far, Reneri has only been evaluated with open-source developers [3] (see Section 2.4.6). We want to assess its effectiveness on undergraduate computer science students, thus shifting the target group from expert developers to apprenticeship students. The comparison baseline we consider is classical mutation testing. Our research questions are:

**RQ1**: To what extent are the results of Oliveira et al. [34] and Delgado-Pérez et al. [35] (i.e., classical mutation testing is effective for teaching the basics of software testing) replicable in our context (i.e., beginner students at the University of Namur)?

**RQ2**: What is the impact of extreme mutation testing on students' unit testing learning?

**RQ3**: Compared to classical mutation testing, to what extent does extreme mutation testing have a larger, smaller, or similar impact on students learning to write unit tests?

The mutation testing tools used in the evaluation are PIT and Reneri. The former uses the Gregor mutation engine (classical mutation operators, see Section 2.4.3), the latter uses the Descartes engine (extreme mutation operators, see Section 2.4.6). We will measure the tests' quality using instruction coverage and mutation score, and use self-evaluation and a questionnaire to measure the impact on learning.

Our hypothesis is that Reneri is clearer for novice testers, as it is more user-friendly (being text-based) and generates "more obvious" mutants (being coarse-grained). Consequently, we hypothesise that Reneri will be more effective than

PIT for undergraduate students. To check this, we designed and carried out an experiment which is described in this chapter.

We planned to conduct the experiment with students taking the "Object Oriented Design and Programming" course at our faculty. This is an introductory course to object oriented programming in Java for undergraduate second-year students in computer science and business engineering. We made this choice because students learning programming and testing are our target audience. In case we need more participants, we also planned to include third-year computer science students taking the "Introduction to Scientific Research Seminar" course. The latter is an introduction to research through practical exercises. This choice is justified because it provides them with an example of an experimental evaluation.

### 3.1.1 Overview

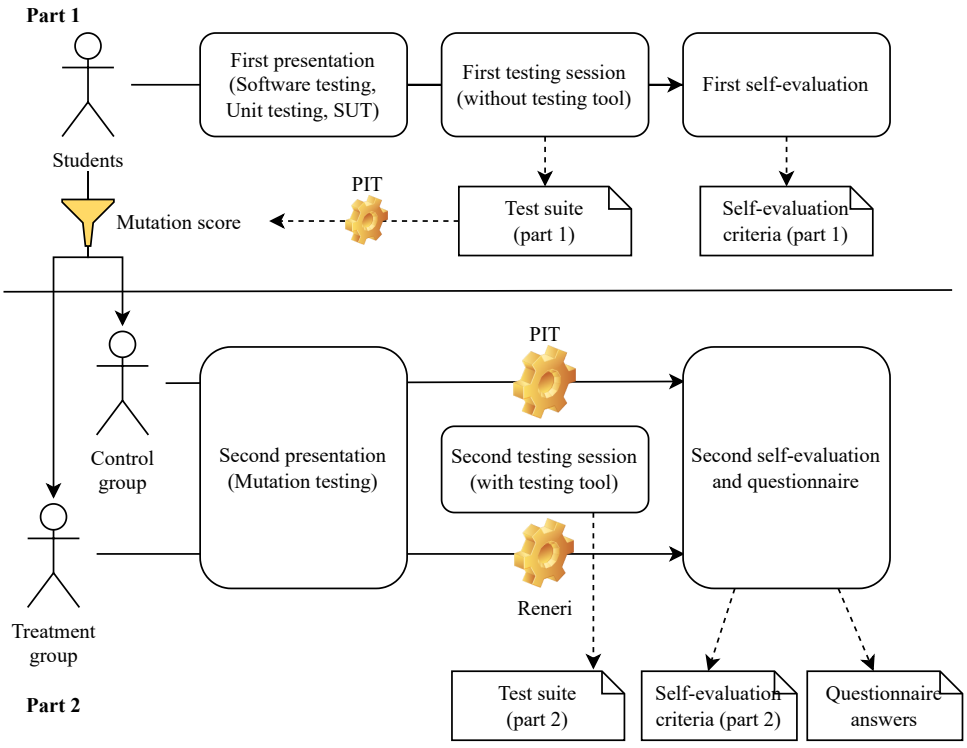



Figure 3.1: Experiment Design

Our goal is thus to determine whether the Descartes/Reneri approach [24, 3] helps undergraduate students in learning software testing and whether it is more or less effective than a traditional mutation testing tool.

The experiment was inspired by that of Delgado-Pérez et al. [35] which itself took inspiration from Oliveira et al. [34]. Testing sessions were organised with undergraduate students. Each session lasted 3 hours and consisted of two parts.

During the first part, students were asked to manually write unit tests on a given system under test (SUT) without any tool. During the second part, they were asked to improve their test suite by adding or editing the test cases they wrote, using a mutation testing tool to help.

Before the first part, there is a presentation where we introduce the students to unit tests. Topics covered include (1) why do we need automated testing, (2) what is a unit test and how to write one, and (3) an applied example. Then, we also present the SUT. We chose and adapted a Java version of the 2048 game. The original implementation can be found on *Rosetta Code*.[1]

Students were split into a control group and a treatment group during the break. In order to preserve a similar distribution of skills, we classified students according to their mutation scores after the first part, and separated similar student pairs into different groups. On the one hand, the control group received a traditional mutation testing tool, PIT, while on the other hand, the treatment group received Reneri.

The experimental process is illustrated in Figure 3.1. We describe it in more detail in the following sections. Finally, a summary of the experiment and the duration of each of its stages can be found in Section 3.1.9.

### 3.1.2 First Presentation

The first presentation was a brief introduction to unit testing and highlighted the advantage of automated testing over manual testing. An example of manual testing is described while we explore its main limitations, i.e., the amount of time it takes to do a manual check-up, the uncertainty of the process, and the need to redo it every time the source code changes.

Next, we show how to make a unit test using the same example. We use the *Arrange, Act, Assert pattern* [7] (or AAA pattern, see Section 2.3). It is presented as a guide to design tests. We recommend that students use it but we do not enforce it. The emphasis is on the similarity of writing such a test compared to a manual one. We insist that they both have the same form, except for the assertion: where we used a print statement in the first place for manual check of the output, a JUnit assertion was used in the latter to automate the check. Moreover, in order to simplify, we only use *assertTrue*, ignoring other JUnit forms of assertion (i.e., *assertFalse*, *assertEquals*, *assertNull*, *assertSame*, etc.).

By insisting on the similarity of automated and manual tests and by presenting only a (very) small subset of JUnit features, we aim to address one challenge previously seen in Figure 2.2: the increased cognitive load when learning a new

[1] It is available at `https://www.rosettacode.org/wiki/2048#Java`.

tool. We hope to give students a positive image of software testing, as it can be accessible and valuable.

### 3.1.3 System Under Test

This introduction was followed by a presentation of the system under test: 2048, a video game by Gabriele Cirulli made in 2014 that quickly became very popular. The goal is to combine tiles numbered from 2 to 2048. The tiles are placed on a 4×4 grid and can be merged when they contain the same number. When two tiles are merged, their numbers are added together. Thus, all tiles are powers of two. The player may move all tiles simultaneously up, down, left, or right. Finally, if no more moves are possible, the game is lost. A screenshot of the implementation we used can be seen in Figure 3.2.

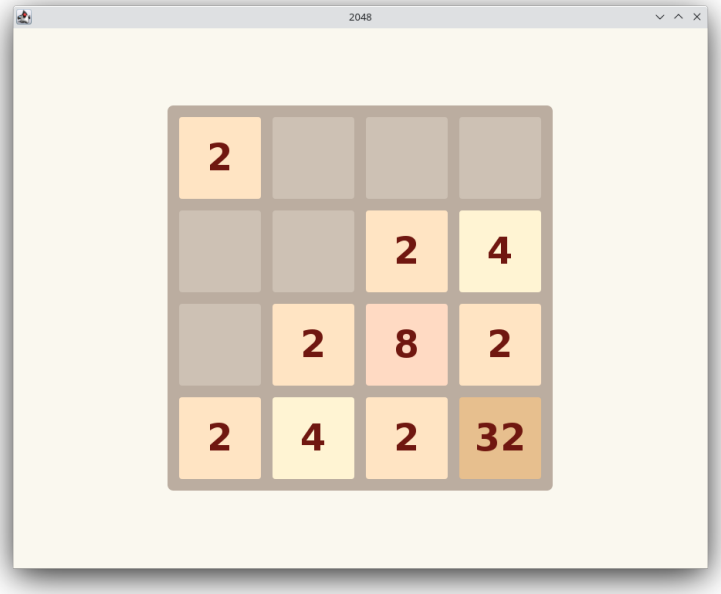



Figure 3.2: An In Progress Game of the Provided 2048 Implementation

2048 has been chosen because it is widely known, and its rules are easy to grasp. In addition, using a game as a SUT is a way to introduce gamification. This was inspired by the *JPacman* framework [30] mentioned earlier. Gamification, both with the ease of use presented above, are ways to overcome the fact that software testing is often not well accepted by students. This is another challenge from Garousi et al. [28] (Figure 2.2).

However, the original 2048 game contains randomness. Indeed, new tiles appear at a random position. They also have a 10% chance of having an initial value of 4 instead of 2. However, as unit tests should be repeatable (the "R" in

FIRST principles, see Section 2.3), we chose to alter the rules and avoid these random behaviours. Thus, new tiles always appear in the first available location in the provided implementation, starting from the top left. Besides, the value of new tiles follows a pattern: the first tile will contain 4, the nine subsequent tiles contain 2, and then the pattern starts over again. That way, we eliminate the probabilistic nature of the original ruleset and simplify the testers' work. The adapted source code is available in the "SUT" folder on our *Zenodo* [36] repository (see Chapter 4).[2]

### 3.1.4 First Testing Session

Students had to write unit tests for the SUT throughout this session until they felt it was enough. They could run the tests whenever they wanted but could not see the code's mutation score or statement coverage. To help novice testers, we suggested that they start with a specific method of a specific class: namely, the *equals* method of the *Tile* class.

After one hour, we asked them to stop and complete a self-assessment of their work based on five criteria inspired by Delgado-Pérez et al. [35], rating themselves from 1 (poor) to 4 (good) for each criterion. Since the students were not expected to be fluent in English, we translated the criteria into French to avoid language-related interpretation problems (English version in Table 3.1). After the assessment, the students were given a short break with refreshments before the second part of the experiment.

### 3.1.5 Group Separation

During the break, students were separated into two groups. In the second testing session, one group will have access to PIT reports and the other to Reneri reports. We used the mutation score to split them equally and have as many students with a high, medium, or low score in both groups. This is based on the idea that the score represents the real competence of the student

In concrete terms, the mutation score was calculated for each test suite. Then the students were sorted in ascending order according to their score. Finally, the first student was selected for group 1, the next for group 2, then again for group 1, and so on.

[2]This repository can be found at `https://doi.org/10.5281/zenodo.6629272`.

Table 3.1: Self-Evaluation Criteria [35]

| Name | Definition |
|---|---|
| **C1. Correctness** | The tests do not cause errors when they are executed on the original program. |
| **C2. Completeness** | The tests represent a sufficiently large portion of the different use scenarios of the program. |
| **C3. Assertions** | The tests contain a sufficient number of assertions that cover all the changes of state of the program due to its execution. |
| **C4. Design** | Each test is designed to verify a specific functionality (i.e., a test does not combine several unrelated usage scenarios) and there are no tests that are redundant or overlapping. |
| **C5. Legibility** | The programming style is clear and makes it possible to understand the purpose of the tests and their assertions. |

### 3.1.6 Second Presentation

The second part began with an introduction to the basics of mutation testing. Reusing an example from the first presentation, we explained that mutation testing consists of artificially introducing "bugs" into the SUT and then checking whether the test suite can detect these bugs. We insisted that at least one test case should fail and that the test suite is inadequate if no test fails.

Since most of the students were novice testers, we tried to avoid overwhelming them with too much information and technical or theoretical considerations. For example, we simplified by using the term "bugs" instead of "defects". Also, issues such as redundant or equivalent mutants (see Section 2.4.2) were not addressed.

### 3.1.7 Second Testing Session

After the break, we asked the students to improve the test suite they wrote in the first session by adding new tests or editing existing ones. However, now they have access to a mutation testing tool, either PIT or Reneri, depending on their assigned group. During this session, the actual experiment took place as it allowed us to compare the evolution of the students according to the tool they used.

We adapted Reneri to translate its reports into French and make it generate HTML directly, whereas it originally made text-only reports. In addition, the content of the reports were clearly separated into three parts: problem, diagnosis,

and solution. These changes aim to adapt the tool from expert to novice programmers while keeping the information from the original reports. The changes are illustrated and described further in Section 3.3.

At the end of the session, we asked the students to self-evaluate their first test suite with the same criteria again (Table 3.1). We stressed that the evaluation was done on the test suite from the first part and not on the one they had just written. Indeed, our objective is to measure the evolution, after introducing a tool, of students' perception of their original test suite written without help.

### 3.1.8 Subsequent Questionnaire

Lastly, we adapted and used the survey from Delgado-Pérez et al. [35] which itself was adapted from Oliveria et al. [34]. We added two questions. The first one (Q1) focuses on Java skills. It was added because some students struggled with Java more than JUnit itself during dry runs (see Section 3.2). Besides, we had students from different educational backgrounds. We wanted to assess that to be possibly able to analyse data based on the students' initial level. The second question we added (Q12) is about the understandability of the generated report (whether by PIT or Reneri). We wanted to compare the two, wondering if a text-based report (Reneri) is more understandable or clearer than a code-based report (PIT).

The actual questions and answers will be presented in Chapter 5 (Table 5.1).

### 3.1.9 Summary

All tasks with their duration are summarised in Table 3.2 below. The experiment lasts 3 hours in total.

Table 3.2: Experiment Steps and Duration

| **Task** | **Duration** |
|---|---|
| Unit testing introduction and SUT presentation | 30 minutes |
| Testing session without tool | 60 minutes |
| First self-evaluation | 5 minutes |
| Break, mutation score generation, and separation into groups | 15 minutes |
| Mutation testing introduction | 10 minutes |
| Testing session with PITest/Reneri | 45 minutes |
| Second self-evaluation | 5 minutes |
| Questionnaire | 10 minutes |

## 3.2 Dry Runs

Dry runs were conducted with three students to test and improve the experience and Reneri's outputs. These took place in a more informal context, but we tried to respect the protocol as closely as possible. Care has been taken to select students who will not participate in the actual experiment.

### 3.2.1 First Student

The first person is a graduate computer science (CS) student with a cybersecurity background. During the first part of the experiment, we observed some confusion between debugging and testing. Indeed, the student felt that his tests were useless because the SUT seemed not to contain any bugs. His mindset was to erase bugs instead of checking if the program worked. It is a commonly reported problem in students' perception of software testing [28]. We also noticed some difficulties with the Java programming language in itself. The student said it had been a while since he had used Java and that he could have gone further if he had not wasted time understanding the source code. Furthermore, the first actual unit test was written after 45 minutes.

During the second testing session, Reneri generated only one hint. The student managed to resolve it. Then, we stopped the experiment as the report was blank because there were no more pseudo-tested methods, and any form of code coverage was not yet implemented in Reneri.

Throughout this first run, we helped the student as little as possible. We wanted to see how he would do on his own. However, to address the difficulties he encountered with Java, we decided at that point to give help to students in the same situation. We would tell them beforehand that we can give them help if they need it, and, during the following experiments, we would give guidance on problems related to programming in Java, but not on the design of unit tests themselves. In this way, we believe we will not bias the results with misunderstandings of Java, nor will we directly give test solutions. In addition, we also refactored the source code a bit to make it more accessible.

Finally, we implemented method coverage into Reneri, thus adding *not-reached* hints on top of *not-infected*, *not-propagated*, and *not-revealed* hints (see Section 2.4.6). *Not-reached* hints would appear only after any other hint types. This choice was made to avoid the blank report we had. We assumed that once solved by adding a test case, *not-reached* hints would lead to more specific Reneri hints.

### 3.2.2 Second Student

We also ran the experiment with an undergraduate business engineer student. He did not have a computer science background except for an introductory course on Java. We first considered including every student taking this course. However, the experiment was quite challenging for him, as he was having trouble understanding the source code. In the end, he did not manage to write a single unit test by himself.

As a result, we decided to focus on computer science students only. We judged that the SUT was too complicated for non-CS profiles and that the results would be useless as part of this experiment.

### 3.2.3 Third Student

A third and final dry run was made with an undergraduate computer science student, representative of our target group, and with modifications described in Section 3.2.1. This student had a good knowledge of Java because he had used it recently in other projects. So, the first testing session went smoothly.

During the second session, the introduction of Reneri led to the generation of a few non-coverage hints. By adding a single test case, the student solved them. Then, and until the end of the time limit, he encountered coverage hints only. He went on writing new tests. We noticed that some more complicated methods were causing problems for him, especially in the *GameController* class.

After this run, we decided to sort *not-reached* hints by class in increasing complexity. Hints about the *Tile* class will be presented first, then those about the *Grid* class. In this way, only the most experienced students will be confronted with the most complex code.

## 3.3 Changes Made to Reneri

In its original version, Reneri produced text-only English reports as seen in Figure 3.3. We wanted to make the reports more acceptable for learning students. This intention is reflected in Figure 3.4. The two figures present the exact same hint.

Reneri only generates JSON files containing hints information and a Python script is actually used to transform that information into a readable report. On one hand, we modified Reneri to also produce information about *not-reached* (i.e., uncovered) methods. On the other hand, we adapted the Python script to (1) directly generate an HTML report, (2) include the information concerning

The body of the method be.unamur.game2048.models.Tile.getNearestPower2(int) was replaced by `return 1;` yet, none of the following tests failed:

- be.unamur.game2048.Test2048.testTileGetValue(be.unamur.game2048.Test2048)
- be.unamur.game2048.Test2048.testTileIsMerged(be.unamur.game2048.Test2048)
- be.unamur.game2048.Test2048.testTileMergeWith(be.unamur.game2048.Test2048)
- be.unamur.game2048.Test2048.testTileEquals(be.unamur.game2048.Test2048)
- be.unamur.game2048.Test2048.testTileCanMergeWith(be.unamur.game2048.Test2048)

It is possible to observe a difference between the program state when the transformed method is executed and the program state when the original method is executed. This difference can be observed in Test2048.java from the expression returning a value of type `be.unamur.game2048.models.Tile` located in line 23 from column 25 to column 26

When the transformation is applied to the method, it was observed that the field `value` of the value obtained from the expression was `1` but should have been `2`.

Consider modifying the test to verify the value of `value` in the result of the expression.

Consider verifying the result or side effects of one of the following methods invoked for the result of the expression:

- `be.unamur.game2048.models.Tile.canMergeWith(be.unamur.game2048.models.Tile)`
- `be.unamur.game2048.models.Tile.toString()`
- `be.unamur.game2048.models.Tile.getValue()`
- `be.unamur.game2048.models.Tile.mergeWith(be.unamur.game2048.models.Tile)`

Figure 3.3: Example of an Original Reneri Report

## `Tile.getNearestPower2(int)`

Le corps de la méthode `Tile.getNearestPower2(int)` a été remplacé par `return 1;`. Pourtant, aucun des tests suivants n'a échoué :

- `Test2048.testTileEquals`
- `Test2048.testTileGetValue`
- `Test2048.testTileIsMerged`
- `Test2048.testTileMergeWith`
- `Test2048.testTileCanMergeWith`

Il est possible d'observer une différence dans l'état du programme selon que l'on utilise la méthode originale ou la méthode modifiée. Cette différence se situe dans `Test2048.java` à la ligne 23 dans une expression de type `be.unamur.game2048.models.Tile`. En effet, la valeur de la variable d'instance `value` était `2` avec la méthode orignale, alors qu'elle vaut `1` avec la méthode modifiée.



Ainsi, une solution pourrait être d'ajouter une assertion ciblant `value`. Voici la liste des méthodes que vous pourriez appeler pour ce faire :

- `Tile.canMergeWith(be.unamur.game2048.models.Tile)`
- `Tile.toString()`
- `Tile.getValue()`
- `Tile.mergeWith(be.unamur.game2048.models.Tile)`

Figure 3.4: Example of an Adapted Reneri Report

uncovered methods, and (3) translate it into French. With the exception of the format and translation, the content of the report remains essentially the same. By doing so, we hope to make the reports more accessible to novice students.

The adaptations we made to Reneri is publicly available on our *Zenodo* [36] repository (see Chapter 4). More specifically, it can be found in the "descartes-reneri" folder.[3]

[3]That can itself be found at `https://doi.org/10.5281/zenodo.6629272`.

# Chapter 4

# Dataset

Following the best practices of open-science, we decided to publicly share the raw results of the experiment and the source codes used during it. This ensures replicability, as we provide a replication package containing the very code we used to compute the data that we present in the following chapter. In addition, it allows readers to take a look at the system under test (see Section 3.1.3) and at the changes we made to Reneri (see Section 3.3). Finally, it represents a contribution to open data, as we are sharing the students' test suites which could be used in other studies on the topic.

The platform we chose to share our data on is *Zenodo* [36]. *Zenodo* is a website tailored for open-science, allowing the deposit of open access archives. It was commissioned by the European Commission in a effort to support its open-data policy. The data described below can therefore be found in our repository at `https://doi.org/10.5281/zenodo.6629272`.

The primary data generated by this experiment are therefore the students' written tests, their questionnaire responses, and their two self-evaluations. The "tests" folder contains the test suite for each student and for both parts. Please note that everything has been anonymised.

From the tests, and using automated scripts, we derived secondary data such as reports found in the directory of the same name. There are three types of reports: reports for statement coverage (from JaCoCo), reports for mutation coverage (from PIT), and reports with hints (from Reneri). Again, the data is generated for each student and for each part, regardless of their assigned group.

Finally, from these reports, we derived tertiary data as *comma-separated values* files (CSV files), such as "coverage.csv" from the JaCoCo reports, "hints.csv" from the Reneri reports, and "mutants.csv" from the PIT reports.

From these CSV files, we generated the tables and plots presented in this document. We used Python in a *Jupyter* [37] notebook using the *pandas* [38, 39]

and *seaborn* [40, 41] libraries. The contents of the different columns in each file are described below.

**Common columns**

- **`id`**: The student's ID, completely anonymous (from "1" to "43").
- **`group`**: The student's group ("pit" or "reneri", see Figure 3.1).
- **`part`**: The session number of the experiment ("1" or "2", see Figure 3.1).

**Answers to the questionnaire (in "questions.csv")**

- **`q1` to `q14`**: The 14 answers to the questionnaire (see Table 5.1).

**Self-evaluations (in "criteria.csv")**

- **`c1` to `c5`**: The five self-evaluation criteria (see Table 3.1).

**Reneri hint types (in "hints.csv")**

- **`not_reached` to `not_revealed`**: The number of hint types related to the RIPR model (see Section 2.4.4).

**Mutation scores (in "mutants.csv")**

- **`total`**: The total number of killed mutants.
- **`Tile` and `Grid`**: The number of mutants killed in the respective class.
- **`returns.BooleanFalseReturnValsMutator`, etc.**: The number of killed mutants of the respective mutator type.[1]
- **`Tile.getNearestPower2` to `Grid.clearMerged`**: The number of killed mutants in the respective method.

**Code coverage (in "coverage.csv")**

- **`class`**: The covered class ("Tile" or "Grid").
- **`instruction_missed` to `method_missed`**: The number of missed nodes according to the respective coverage metric.[2]
- **`instruction_covered` to `method_covered`**: The number of covered nodes according to the same metrics.

[1] Refer to `https://pitest.org/quickstart/mutators/`.
[2] Refer to `https://www.jacoco.org/jacoco/trunk/doc/counters.html`.

# Chapter 5

# Results

## 5.1 Participants

We ran the experiment at the computer science faculty at the University of Namur (UNamur, in Belgium). It was run twice, the first time with 26 second-year undergraduate students and the second time with 17 third-year students, which makes a total of 43 students. The students were split equally into groups based on their results after the first testing session: 21 in the control group (PIT, classical mutation testing) and 22 in the treatment group (Reneri, extreme mutation testing).

Based on their answers to the questionnaire (Table 5.1, Q1, Q2, and Q7), 28 (68%) of them had a no prior or only basic knowledge of Java, and 13 (32%) had more than intermediate or advanced knowledge. The high proportion of beginners can be attributed to the fact that they are still undergraduates. Among them, 32 (79%) had no prior knowledge about software testing or had only basic knowledge, while 9 (22%) had intermediate or advanced knowledge. Indeed, in their curriculum, the second-year students had not yet attended a lecture teaching unit testing concepts. As for third-year students, they were introduced to these concepts in two different programming courses, neither of which focuses exclusively on software testing. This could explain why they feel that software testing is not sufficiently or not at all introduced during their studies, as 27 (64%) students stated in Q7.

In a perfect world, complete data would be available for each participant. However, this is not the case, as some students submitted a failing test suite, giving blank values in datasets, and others did not complete the self-evaluation for one or more criteria. We adopted a fairly standard approach to deal with missing data [42] by excluding them from the analysis when there was an empty value for one or both parts for a given metric. Therefore, we were left with groups

of 16 and 15 students for mutation scores, coverage, and hint types, and with groups of 20 and 19 students for self-evaluation criteria (down to 18 for C4 and 17 for C5 in the second group).

## 5.2 Hypothesis Testing

We defined three research questions in Chapter 3. They are repeated here as a reminder. The way in which we will answer these questions is illustrated in Figure 5.1 below. It illustrates the link between the research questions and the metrics used.

**RQ1**: To what extent are the results of Oliveira et al. [34] and Delgado-Pérez et al. [35] (i.e., classical mutation testing is effective for teaching the basics of software testing) replicable in our context (i.e., beginner students at the University of Namur)?

**RQ2**: What is the impact of extreme mutation testing on students' unit testing learning?

**RQ3**: Compared to classical mutation testing, to what extent does extreme mutation testing have a larger, smaller, or similar impact on students learning to write unit tests?

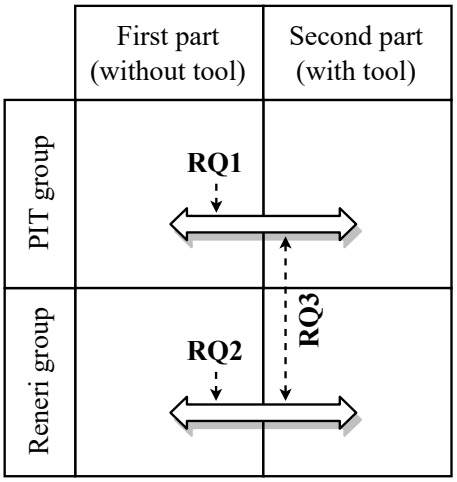



Figure 5.1: RQ1 evaluates the evolution between the first and second part for students who used PIT. RQ2 does the same thing but for Reneri users. RQ3 compares the evolution of the PIT group to that of the Reneri group.

In order to validate our different hypotheses, we used two non-parametric rank tests depending on whether the observations were independent or not. On the

one hand, we are comparing groups to each other (RQ3). As they are independent, we used a *Mann-Whitney U test* [43] (we also provided the *U-value* for this test). On the other hand, we are observing the evolution of the same group across the two parts (RQ1 and RQ2). In that case, the observations are dependent, so we used a *Wilcoxon signed-rank test* [44] (we also provided the *W-value* for this test). We used the implementations from Python's *scipy* library [45]. Finally, we used the fairly standard value of 0.05 for the *p-value* threshold.

To interpret the effect size, we will use *Vargha-Delaney's* $\hat{A}_{12}$ [46]. It measures the stochastic dominance, i.e., how often, on average, the treatment technique outperforms the control one. For instance, an $\hat{A}_{12} = 0.7$ means that there is a 70% chance that a random student from the treatment group outperforms a random one from the control group. Consequently, if $\hat{A}_{12} = 0.5$, it means both techniques are equal. If $\hat{A}_{12} < 0.5$, the control outperforms the treatment. And $\hat{A}_{12} > 0.5$ means the opposite. We will use the rule of thumbs given in the original paper, i.e., $\hat{A}_{12} > 0.56$ or $\hat{A}_{12} < 0.44$ is *small*, $\hat{A}_{12} > 0.64$ or $\hat{A}_{12} < 0.36$ is *medium*, and $\hat{A}_{12} > 0.71$ or $\hat{A}_{12} < 0.29$ is *large*.

## 5.3 Mutation Score

Mutation score is the first metric we analysed to measure the test suite's quality. It was computed for each student at the end of the first and second testing sessions (i.e., before and after introducing the mutation testing tool). This allowed us to compare the evolution according to the tool in use. Tables containing the full data can be found in appendix (Tables A.1 to A.6). These tables contain the ratio of mutants killed (i.e., mutation score) in total, in each class, and in each method, as represented by the mean, the first quartile, the median, the third quartile, and the interquartile range (IQR). Finally, these data are illustrated in Figure 5.2.

In regards to RQ1, we see that students using PIT during the second part quite significantly improved their test suite's mutation score ($W = 0.00$, $p < .001$). This might have been expected simply because they received extra time to write unit tests. So, it is more interesting to look at the effect size. In this case, it can be considered *large* ($\hat{A}_{12} = 0.84$). This observation is consistent with the works of Oliveira et al. [34] who showed that mutation testing is beneficial to learning for novice programmers.

In regards to RQ2, we observed that students also improved their test suite. The same statistical test was performed and this improvement was shown to be quite significant as well ($W = 0.00$, $p = .001$). However, this time, the effect size was found to be *medium* ($\hat{A}_{12} = 0.66$).

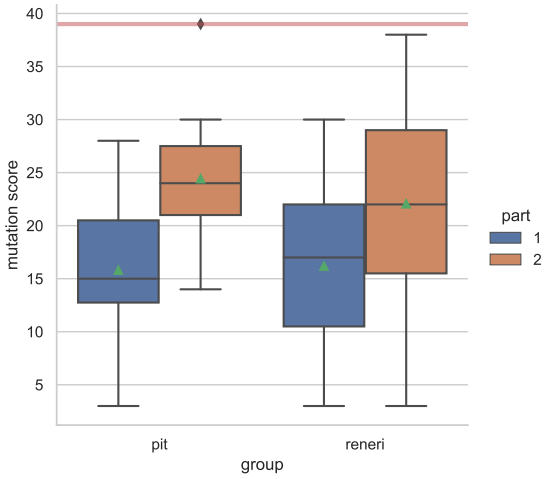



Figure 5.2: Evolution of the distribution of killed mutants at the end of the first and second part for each group. The red line represents the total number of killable mutants. Green triangles represent the means.

In regards to RQ3, we have just seen that PIT has a larger effect size than Reneri. At first glance, the mutation score differences between the two sessions seem to rather be lower for Reneri users compared to PIT users ($U = 160.50$, $p = .057$, with a *medium* effect size of $\hat{A}_{12} = 0.33$). However, we must be cautious: this result is not as clear-cut as the previous ones, as we have a much higher *p-value*, slightly above the threshold. We cannot conclude that there is a significant difference.

However, if we look at the interquartile ranges (visually on Figure 5.2 and in raw numbers in Table A.6), we see that there is more variability in the results for Reneri users. Indeed, for PIT users the IQR goes from 7.8 to 6.5 mutants ($\Delta = -1.2$) while for Reneri users it goes from 11.5 to 13.5 ($\Delta = +2.0$).

To further understand what is going on, we have separated the results according to the two classes in the SUT's source code, i.e., *Tile* and *Grid* (in the same order the students were told to test them). This separation can be seen in Figure 5.3.

On the one hand, as far as the *Tile* class is concerned, we see that the students' mutation scores improved significantly more for PIT users than for Reneri users ($U = 225.00$, $p < .001$, with a *large* effect size of $\hat{A}_{12} = 0.06$). The *p-value* indicates that this result is much stronger for this specific class than for the mutation score of both classes combined.

On the other hand, if we focus solely on the *Grid* class, it seems that Reneri users improved their mutation score more than PIT users. However, the *Mann-Whitney U test* failed to reveal that it is actually the case ($U = 103.00$, $p = .223$, with a *small* effect size of $\hat{A}_{12} = 0.57$). Therefore, we cannot clearly draw a conclusion for this case.

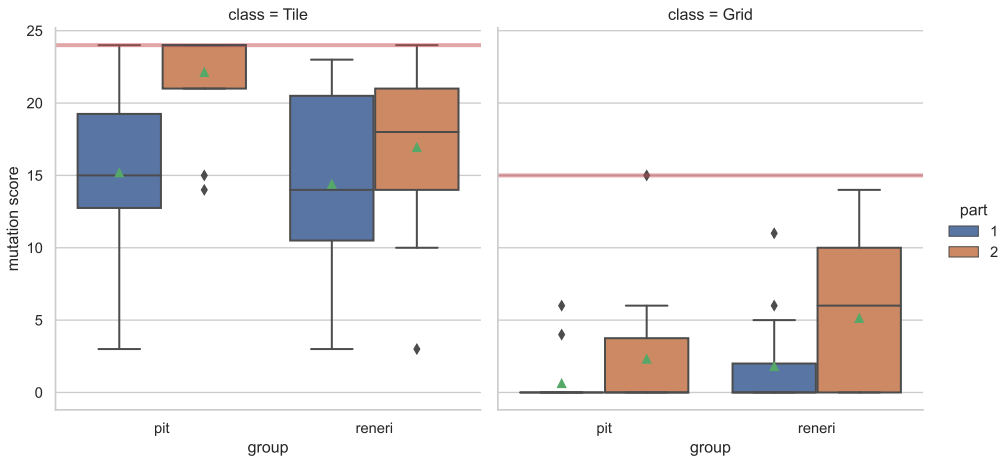



Figure 5.3: Evolution of the distribution of mutation scores per class at the end of the first and second part for each group. Red lines represent the total number of killable mutants. Green triangles represent the means.

By further looking at mutation scores method by method (see Figure A.1 in appendix, the methods are sorted in the same order as in the source code), we can indeed see that Reneri users seem to have made progress on more methods than PIT users, especially regarding the *Grid* class.

Simply by looking at the graph, however, we see a difference concerning the mutation scores after the first testing session and before the second one. The third quartiles confirm this by showing us that only a few outliers of the PIT group started testing this class during the first part ($Q3 = 0$, Table A.5) whereas a quarter of the Reneri group had already started testing it then ($Q3 = 2$, Table A.5).

Therefore, since the statistical test failed to reject the null hypothesis, and since there seems to be an imbalance between the PIT and Reneri groups from the start, we must bear in mind that any result concerning the *Grid* class should be taken with caution.

## 5.4 Instruction Coverage

The previous observation shows that Reneri could thus lead to better coverage than PIT. To verify this, we calculated the code coverage of each test suite using the *JaCoCo* coverage library. Detailed numbers can be found in appendix (Tables A.7 and A.8). They are reported on Figure 5.4 which shows the evolution of instruction coverage per class and for each group.

Concerning the *Tile* class, PIT users clearly improved their instruction coverage more than Reneri users ($U = 171.50$, $p = .020$, with a *medium* to *large* effect size of

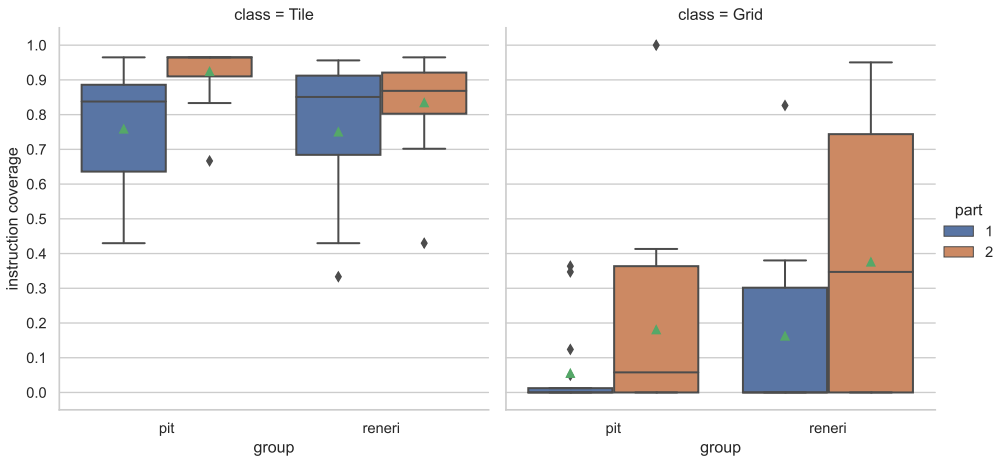



Figure 5.4: Evolution of the instruction coverage ratio for each class before and after the introduction of the tool. Means are represented as green triangles.

$\hat{A}_{12} = 0.29$, being just at the threshold). For the *Grid* class, we fail to prove that the Reneri group performed significantly better ($U = 112.50$, $p = .379$). This can be attributed to the fact that a part of this group had already started testing this class during the first testing session, unlike the other group. This left them with an initial advantage. Even if it had been proven, the effect size seems insignificant ($\hat{A}_{12} = 0.53$). However, we still cannot draw any conclusion. These observations are consistent with the previous section (Section 5.3).

## 5.5 Self-Evaluation

As explained in Sections 3.1.4 and 3.1.7, students were asked to rate the tests they wrote during the first session from 1 (poor) to 4 (good) on five criteria, namely *Correctness*, *Completeness*, *Assertions*, *Design*, and *Legibility* (see Table 3.1 for their definitions). This rating took place once after the first testing session and again after the second one. It allowed us to compare the evolution of the students' perception of their work depending on the tool in use. Figure 5.5 presents the evolution of the ratings. We will mainly focus on C2 and C3, as we can see interesting patterns concerning these two. Refer to Figure A.2, Table A.9, and Table A.10 in appendix for the evolution of the other criteria.

On the one side, on average, students slightly increased their evaluation of the *Completeness* criterion (C2) with PIT while it remained at a comparable level for students using Reneri. This criterion is defined as the test suite's coverage of all testing scenarios. With regards to RQ1, its increase in the PIT group means that students eventually felt they were more complete than they had initially

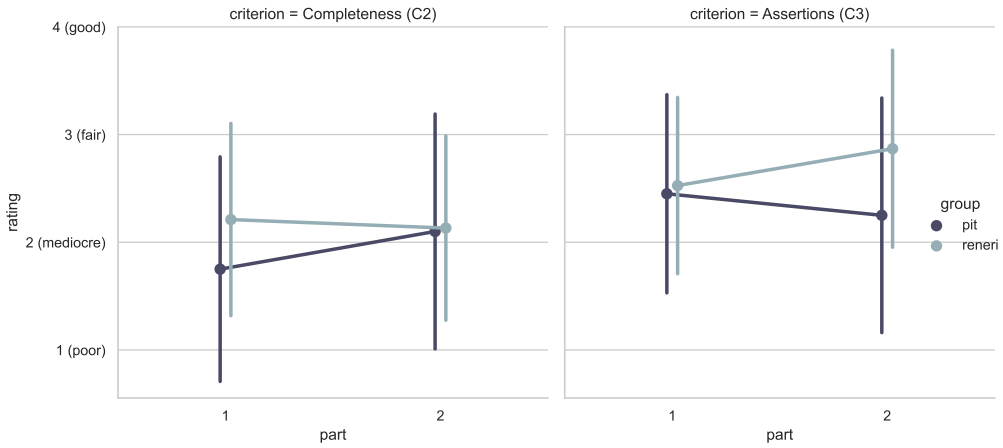



Figure 5.5: Evolution of the C2 and C3 self-evaluation criteria by tool. Points represent means. Vertical bars are confidence intervals covering standard deviations.

thought. On the contrary, for RQ2, the Reneri group did not significantly change its evaluation of the same criterion.

On the other side, the *Assertions* criterion (C3) is defined as the level of adequacy of assertions in the test cases. A good level implies that all the changes in the program state were covered. While C2 evaluates global coverage of the SUT, C3 instead evaluates the strength of individual test cases. Regarding RQ1, we see that C3 decreased with PIT. Regarding RQ2, we see that it increased with Reneri. This means that the first group became aware of some limitations of their original test cases, whereas the latter had the opposite effect.

Finally, to complete our answer to RQ1, we compared our results from the PIT group with Delgado-Pérez et al. [35], from whom the evaluation criteria were taken. They ran a similar experiment where students created tests using mutation testing on a C++ SUT. The authors wrote that "the knowledge of the mutation score significantly decreased the expectations the students had on the test suites, especially concerning their coverage of functionalities and possible changes in the internal state of the program". Indeed, they observed that C2 and C3 both decreased when students used classical mutation testing. We got similar results for C3 but observed the opposite trend for C2, which increased instead. This could be due to using a different SUT with a different level of complexity. Delgado-Pérez et al.'s [35] SUT is described as two C++ classes, but its source code is not public. However, even if one was negative and the other positive, both results indicate that mutation testing impacts C3.

The other criteria (C1, C4, and C5) showed a similar evolution in both groups and are on par with the previous study's results. This is expected, as the tools should impact coverage (C2) and assertions (C3) but not correctness (C1), design

(C4), nor legibility (C5).

## 5.6 Questionnaire

Table 5.1 contains the questions and answers from the questionnaire that is given at the end of the experiment. In this table, we present the responses for all students, for the PIT group, and for the Reneri group separately. The students received a French translation, but we translated it back to English here.

Table 5.1: Questions and Answers by Group

| | **Both** | **PIT** | **Reneri** |
|---|---|---|---|
| *Q1. What was your knowledge of Java before you started this experiment?* | | | |
| No knowledge | 1 (2%) | 0 (0%) | 1 (5%) |
| Basic knowledge | 27 (66%) | 14 (67%) | 13 (65%) |
| Intermediate knowledge | 11 (27%) | 5 (24%) | 6 (30%) |
| Advanced knowledge | 2 (5%) | 2 (10%) | 0 (0%) |
| *Q2. What was your knowledge of software testing before you started this experiment?* | | | |
| No knowledge | 8 (20%) | 4 (19%) | 4 (20%) |
| Basic knowledge | 24 (59%) | 12 (57%) | 12 (60%) |
| Intermediate knowledge | 8 (20%) | 4 (19%) | 4 (20%) |
| Advanced knowledge | 1 (2%) | 1 (5%) | 0 (0%) |
| *Q3. Do you think it is interesting to present the concepts of mutation testing together with the basics of programming?* | | | |
| Yes | 17 (41%) | 9 (43%) | 8 (40%) |
| Yes but superficially | 20 (49%) | 10 (48%) | 10 (50%) |
| No | 4 (10%) | 2 (10%) | 2 (10%) |
| *Q4. What could be the consequences of the use of mutation testing by novice programmers?* | | | |
| Better programs | 10 (24%) | 3 (14%) | 7 (35%) |
| More competent programmers | 6 (15%) | 2 (10%) | 4 (20%) |
| Better programs and more competent programmers | 24 (59%) | 15 (71%) | 9 (45%) |
| Neither | 1 (2%) | 1 (5%) | 0 (0%) |
| *Q5. Do you consider classical testing tools (JUnit) to be useful for teaching programming fundamentals?* | | | |
| Yes | 21 (51%) | 10 (48%) | 11 (55%) |
| Yes, but only with basic functionality | 17 (41%) | 9 (43%) | 8 (40%) |
| No | 3 (7%) | 2 (10%) | 1 (5%) |
| *Q6. Do you consider mutation testing tools to be useful for teaching the fundamentals of programming?* | | | |
| Yes | 15 (37%) | 7 (33%) | 8 (40%) |
| Yes, but only with basic functionality | 21 (51%) | 12 (57%) | 9 (45%) |
| No | 5 (12%) | 2 (10%) | 3 (15%) |

*Continued on next page*

| | Both | PIT | Reneri |
|---|---|---|---|
| *Q7. Considering your background so far (without taking this presentation into account), you feel that the concepts of software testing have been:* | | | |
| Fairly well presented | 14 (34%) | 8 (38%) | 6 (30%) |
| Insufficiently presented | 26 (63%) | 13 (62%) | 13 (65%) |
| Not presented | 1 (2%) | 0 (0%) | 1 (5%) |
| *Q8. Do you think that using software testing tools for learning purposes could be useful for creating good programming habits?* | | | |
| Yes | 41 (100%) | 21 (100%) | 20 (100%) |
| No | 0 (0%) | 0 (0%) | 0 (0%) |
| *Q9. Do you think that creating test cases through mutation testing is useful for improving the learning ability of novice programmers?* | | | |
| Yes | 39 (95%) | 21 (100%) | 18 (90%) |
| No | 2 (5%) | 0 (0%) | 2 (10%) |
| *Q10. How did you find creating tests manually, without the help of a tool?* | | | |
| Easy in general | 13 (32%) | 8 (38%) | 5 (25%) |
| Difficult, especially with regard to the completeness of my tests (sufficient code coverage) | 20 (49%) | 9 (43%) | 11 (55%) |
| Difficult, especially to follow a logical order in the design of test cases | 8 (20%) | 4 (19%) | 4 (20%) |
| *Q11. What is your perception of software testing after applying mutation testing to your tests?* | | | |
| It has changed the way I design tests | 6 (15%) | 2 (10%) | 4 (20%) |
| This allowed me to discover parts of the code that were not sufficiently tested | 30 (73%) | 15 (71%) | 15 (75%) |
| The mutants do not seem to me to be particularly useful for improving the quality of my tests | 5 (12%) | 4 (19%) | 1 (5%) |
| *Q12. The reports generated by the tool used in the second session:* | | | |
| Were sufficiently understandable | 39 (95%) | 20 (95%) | 19 (95%) |
| Lacked comprehensibility but were still usable | 2 (5%) | 1 (5%) | 1 (5%) |
| Were not understandable enough to be usable | 0 (0%) | 0 (0%) | 0 (0%) |
| *Q13. Compared to your original self-assessment, you feel:* | | | |
| You have assessed yourself correctly | 26 (63%) | 13 (62%) | 13 (65%) |
| You have overestimated yourself | 11 (27%) | 6 (29%) | 5 (25%) |
| You have undervalued yourself | 4 (10%) | 2 (10%) | 2 (10%) |
| *Q14. From a practical point of view, mutation testing:* | | | |
| Is very useful | 32 (78%) | 17 (81%) | 15 (75%) |
| Is very useful but not comfortable to use | 8 (20%) | 4 (19%) | 4 (20%) |
| Does not compensate for the effort required to use it | 1 (2%) | 0 (0%) | 1 (5%) |

A total of 21 students (51%) think that using classical testing tools (like JUnit) helps to teach basic programming concepts (Q5), while only 15 students (37%) felt the same way about mutation testing tools (Q6). On the other side, 17 (41%) students think that using classical testing tools only with basic functionality is helpful (Q5), and 21 (51%) students find mutation testing with basic functionality helpful (Q6). Thus, on average, the same proportion of students think that using classical

or mutation testing tools, at least partially, is helpful for learning. However, they mostly feel that mutation testing should be kept to basic functionality. This might be due to the increased cognitive load of such a tool for novice programmers [28].

Students were unanimous (100%) that using testing tools for learning can create good programming habits (Q8). Almost all students (95%) think that designing test cases through mutation testing helps improve the learning capacity of novice programmers (Q9). Hence, they recognise that they have learned something by using the tools. However, in the PIT group, 18 students (86%) think it leads to better programs, and 17 students (81%) think it leads to better programmers. While in the Reneri group, 16 students (80%) think it leads to better programs, and 13 students (65%) think it leads to better programmers (Q4). This could mean that some students felt that Reneri could only improve the test suite but that they would get nothing out of it in terms of learning.

From a practical point of view, 32 students (78%) think that mutation testing is beneficial, and 8 students (20%) think it is useful yet not comfortable to use (Q14). Finally, concerning understandability, both groups (39 students, 95%) equally think that the reports were sufficiently understandable (Q12). We were expecting a difference between code-based reports from PIT and text-based reports from Reneri, but there seems to be none in that regard.

## 5.7 Findings

We can summarise our findings from this chapter in regards to each research question. Based on mutation scores, instruction coverage, self-evaluation ratings, and the responses to the questionnaire, we have drawn the following conclusions:

**RQ1**: Classical mutation testing has a large effect on improving the students unit testing skills. This is on par with Oliveira et al. [34]. On the one hand, PIT users found shortcomings in the assertions of the tests they had written without the tool. In addition, most students feel that they have learned something using PIT. On the other hand, they also felt that their test suite was more complete than they initially thought. Delgado-Pérez et al. [35] observed the same effect concerning assertions but the opposite one for completeness.

**RQ2**: Extreme mutation testing only has a medium impact on students' unit testing skills. Reneri did not change the students' self-evaluation ratings of the completeness of their test suites. However, it did increase the confidence they had in the assertions written prior to the introduction of the tool.

**RQ3**: Classical mutation testing would rather help students improve their mutation score more than extreme mutation testing. On the other side, extreme mutation testing *could* help to improve instruction coverage more than classical mutation testing. The questionnaire shows that, compared to PIT, a slightly smaller amount of students found Reneri useful for learning. On the contrary, the same number of students in both groups found the reports sufficiently clear, suggesting that there is no difference between code-based and text-based reports in this respect.

# Chapter 6

# Discussion

## 6.1 Learning with Mutation Testing

This first section is based upon the results from Chapter 5 and from a student's perspective.

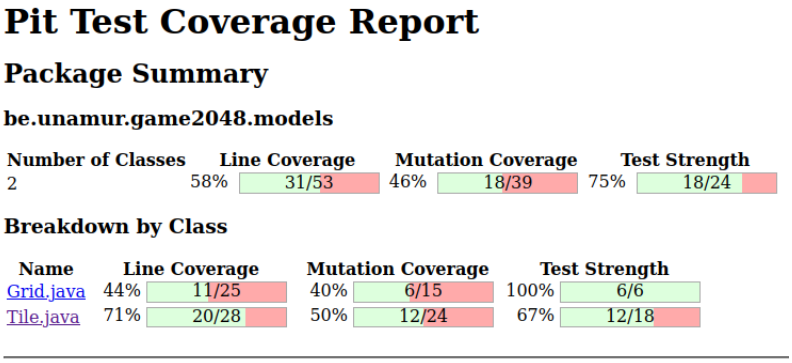
### 6.1.1 A Comparison of the Tools

PIT seems to be more effective than Reneri to improve students' skill set. To discuss this initial finding, we must first have a look at reports generated by both tools. Figure 6.1 is a code-based PIT report, and Figure 6.2 a text-based Reneri report.

As we can see, PIT reports consist of a main page on which global and per class mutation scores can be found (see Figure 6.1a). By selecting a class, one can browse its source code. Lines where a mutant is still alive are marked in red, and line where all mutants are killed are marked in green (see Figure 6.1b). Finally, by hovering a line number, one can see specific mutants (see Figure 6.1c). This kind of report resembles those found in coverage testing tools.

On the other side, Reneri generates textual reports. Figure 6.2 is divided in three parts: *issue*, *diagnosis*, and *solution*. The first part, the *issue* part, describes that removing the entire body of *Tile.equals* and replacing it with "`return false;`" did not provoke any failure. Reneri detected that two tests (*testTileEquals* and *testGridd*) did not fail, yet they called that method. In the *diagnosis* part, the report explains that there were no observable difference whether the original program or the mutated one was executed. Then, in the *solution* part, Reneri suggests creating a variant of one of the two test cases to produce a different return value for *Tile.equals*.

A significant difference between the two reports is that the first one directly



# Pit Test Coverage Report

## Package Summary

### be.unamur.game2048.models

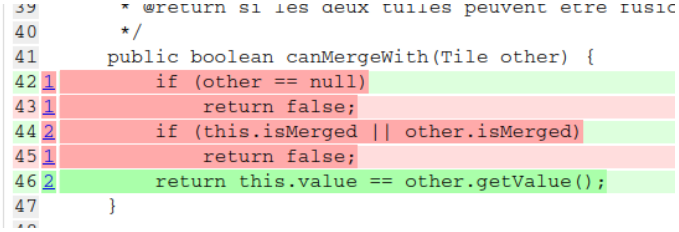
| Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| 2 | 58% | 31/53 | 46% | 18/39 | 75% | 18/24 |

### Breakdown by Class

| Name | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| Grid.java | 44% | 11/25 | 40% | 6/15 | 100% | 6/6 |
| Tile.java | 71% | 20/28 | 50% | 12/24 | 67% | 12/18 |

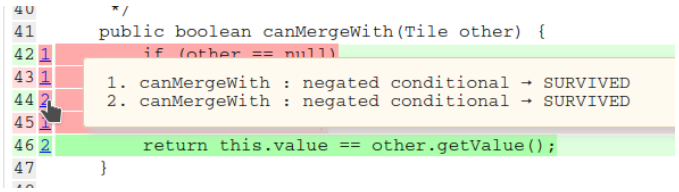
Report generated by PIT 1.7.4

(a) Main Page



```
40        */
41       public boolean canMergeWith(Tile other) {
42 1          if (other == null)
43 1              return false;
44 2          if (this.isMerged || other.isMerged)
45 1              return false;
46 2          return this.value == other.getValue();
47       }
```

(b) Specific Method

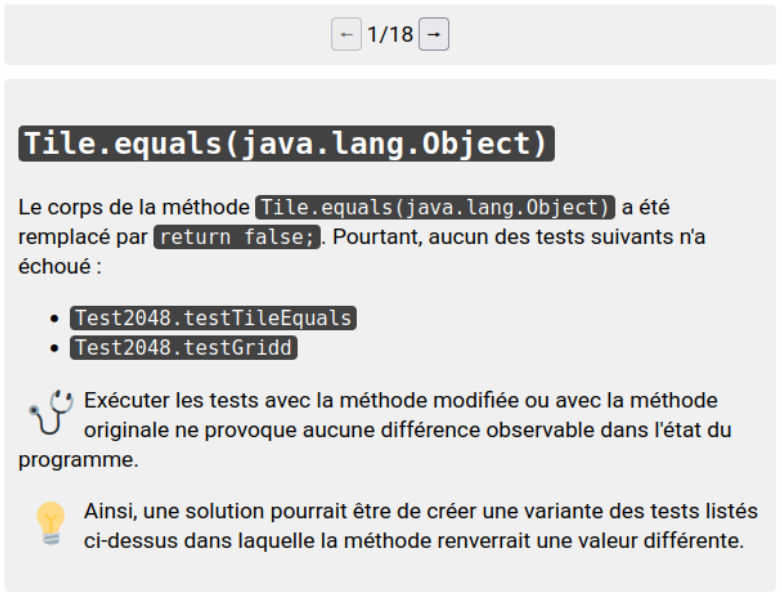


```
41       public boolean canMergeWith(Tile other) {
42 1          if (other == null)
43 1
44 2
45 1
46 2          return this.value == other.getValue();
47       }
```

1. canMergeWith : negated conditional → SURVIVED
2. canMergeWith : negated conditional → SURVIVED

(c) Live Mutants

Figure 6.1: Student 35 PIT Report



← 1/18 →

**`Tile.equals(java.lang.Object)`**

Le corps de la méthode `Tile.equals(java.lang.Object)` a été remplacé par `return false;`. Pourtant, aucun des tests suivants n'a échoué :

- `Test2048.testTileEquals`
- `Test2048.testGridd`

Exécuter les tests avec la méthode modifiée ou avec la méthode originale ne provoque aucune différence observable dans l'état du programme.

Ainsi, une solution pourrait être de créer une variante des tests listés ci-dessus dans laquelle la méthode renverrait une valeur différente.

Figure 6.2: Student 35 Reneri Report

gives line coverage, mutation score, and test strength.[1] This difference could explain the bigger impact PIT has on student's tests mutation scores than Reneri. PIT students have direct access to their mutation score, making it easier to improve it. In addition, PIT, which is fine-grained (traditional mutation operators), shows specific mutants and their exact location in code, in contrast with Reneri, which is coarse-grained (extreme mutation operators). As for Reneri, we will see in Section 6.1.2 that students were mainly encouraged to test uncovered methods instead of improving existing tests.

A second notable difference between the two tools is that Reneri is injunctive whereas PIT is informative. Indeed, Reneri describes a surviving mutant and makes suggestions for adding a new test case or assertion to kill it. PIT only describes mutants and whether they were killed or not. In the first case, the user is told what to do. In the second, they implicitly understand what to do. And, again, as we'll see in Section 6.1.2 below, Reneri users were mostly told to cover uncovered methods.

### 6.1.2 Reneri Hint Types

During dry runs and the actual experiment, we were under the impression that Reneri users were primarily confronted with *not-reached* hints. We even noticed that some users did not have any other type of hints. The problem is that *not-reached* hints are, in fact, only method coverage hints. We hypothesised that the users were biased due to the low number of pseudo-tested methods in their test suites, as Reneri only makes suggestions for pseudo-tested methods.

To verify this, Reneri was run on every single submitted test suite. Then generated hints were classified according to their type: *not-reached*, *not-infected*, *not-propagated*, or *not-revealed*. Figure 6.3 contains the resulting distribution.

The results reveal that, except for a few outliers, no students received *not-infected*, *not-propagated*, or *not-revealed* hints (i.e., extreme mutation hints). Thus, most students only received *not-infected* hints (i.e., method coverage hints). Hence, it is as if most of the students in Reneri's group had only used a coverage method tool, and not a mutation testing tool. Figure 6.4 shows an example of a *not-infected* hint: it says, in French, that no test executed this method, thus simply suggesting to write a new test to cover it.

Out of 43 students' test suites, 28 (65%) did not produce any mutation-related hint. Ignoring coverage, only 4 students (9%) had one hint, 3 students (7%) had two hints, and 1 student (2%) had three hints. Consequently, this gives us a

[1] PIT defines "Test Strength" the same way as mutation score with the difference that it ignores uncovered mutants.

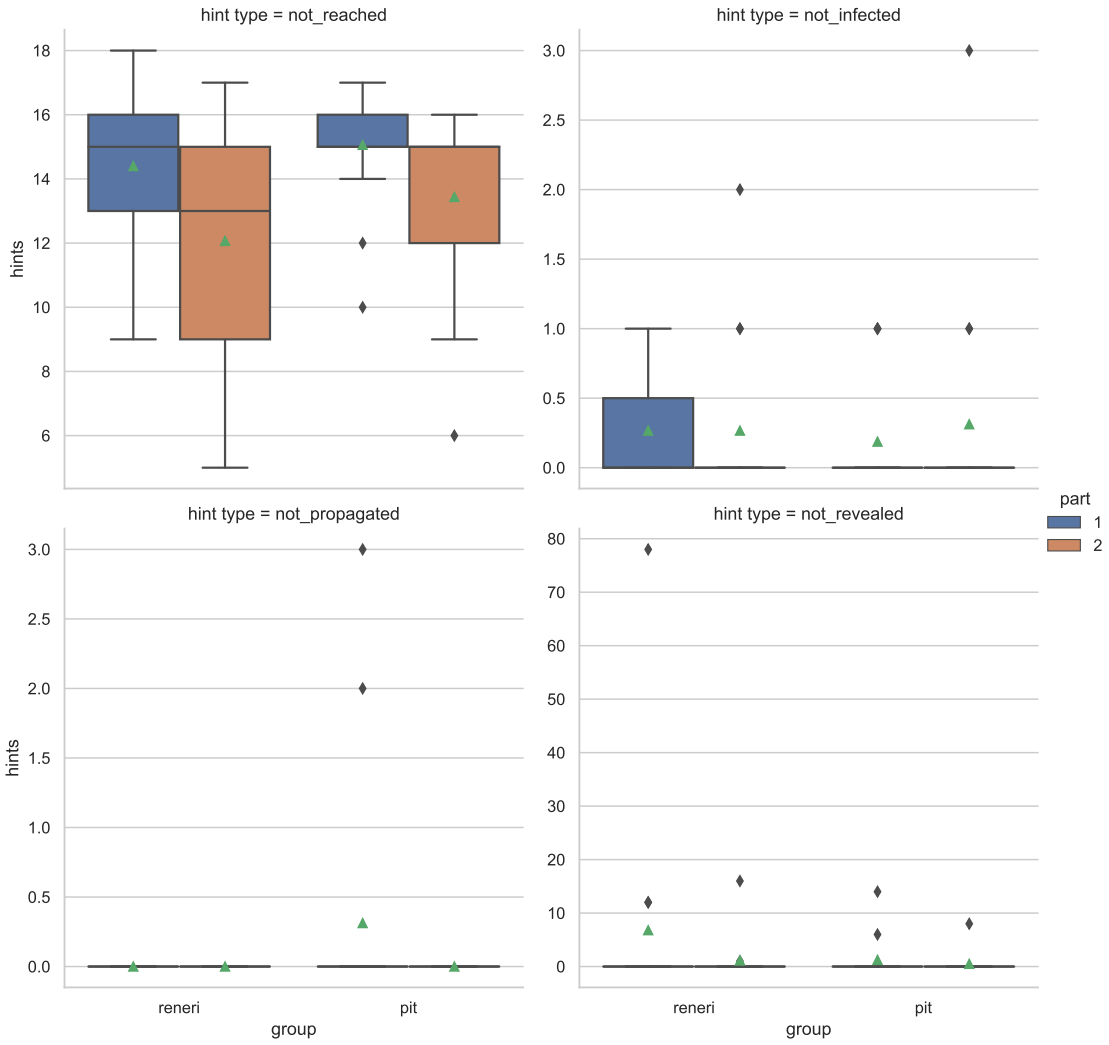



Figure 6.3: Distribution of Reneri Hint Types


Figure 6.4: Example of a *Not-Reached* Hint

new point of view on the results we obtained. In our case, extreme mutation testing did not produce a satisfactory number of mutants that were left unkilled by students. Therefore, Reneri placed great emphasis on method coverage by providing feedback and suggestions mostly focused on this criterion.

Another point came to our attention: the analysis reveals that some test suites generate many *not-revealed* hints. For instance, student 36 generated 78 hints of that type at the end of the first testing session. As for the others, students 6, 31, and 32 generated 14 or 12 hints. Almost every other student generated none. We thus explored the corresponding test suites and generated reports.

Concerning student 36, we noticed that 77 of the *not-revealed* hints concern the same method, *Tile.getNearestPower2*. This method is called within *Tile*'s constructor and ensures that every tile will have a value that is a power of two. Two extreme mutants were generated, the first changed the body of the method to "`return 1;`", the second to "`return 0;`", effectively changing the initialization of the tiles' value. An example test case is presented on Listing 6.1. We can see that the student created two tiles, "`v1`" and "`v2`", and then compared their values. After the mutations, these values were changed to 1 or 0, yet the test did not fail. Indeed, student 36 never wrote any assertion concerning a tile's value. Reneri detected this on lines 4 and 5 and reported 2 hints for both lines, one for each mutant. Reneri also noticed a difference on line 8. More specifically, it detected that the value of the expressions "`v1.getValue()`" and "`v2.getValue()`" changed. In the same expression, it also noticed that "`v1`" and "`v2`" themselves had changed. Reneri thus reported 4 observable differences for each mutant and for line 9 as well. This explains why a single test case produced 20 different *not-revealed* hints. During the second testing session, student 36 added assertions and all the hints were solved simultaneously.

Listing 6.1: Student 36 Example Test Case

```
1   @Test
2   public void testTileGetValue() {
3       // Arrange
4       Tile v1 = new Tile(2);                          // 2 hints
5       Tile v2 = new Tile(4);                          // 2 hints
6
7       // Act
8       boolean check = v1.getValue() == v2.getValue();  // 8 hints
9       boolean check2 = v1.getValue() == v1.getValue(); // 8 hints
10
11      // Assert
12      assertTrue(check2 == true);
13  }
```

By looking at the other students' test cases, we see that they encountered the same situation. Their high number of *not-revealed* hints is also due to combinatorial explosion.

### 6.1.3 A Look Back at the Results

With this new perspective in mind, we can interpret some results of Chapter 5. On the one hand, PIT users had direct insights on the mutants left alive after the execution of their tests. On the other hand, Reneri users were mostly instructed to cover uncovered methods.

#### Mutation Score and Instruction Coverage

In Sections 5.3 and 5.4, we saw that, in terms of mutation score and instruction coverage, the PIT group globally improved more than the Reneri group. This can be explained if we consider that PIT students used classical mutation testing while most Reneri students mainly used method coverage. Indeed, the first group had their mutation score at their disposal as well as the exact surviving mutants to kill to improve this score. The latter did not. Consequently, we can expect that PIT's reports make killing mutants easier.

Students were told to test two classes: first *Tile*, then *Grid*. By grouping by class, we noticed that the PIT group killed more mutants in the first class than the Reneri group but the Reneri group killed more mutants in the second one. It seemed that Reneri users went on testing this second class after finishing testing all methods in the first one. Again, this could be explained by the fact that Reneri reports mostly focused on uncovered code, whereas PIT reports have led to a focus on improving existing test cases. However, we also noticed that the initial mutation score and instruction coverage were actually lower in the PIT group than in the Reneri group, even before part 2 (i.e., before introducing any mutation testing tool). This means that the first group tested *Grid* more than the latter before any mutation testing tool was introduced ($Q2$ = 40% for Reneri, $Q2$ = 0% for PIT, Table A.4).

Concerning instruction coverage, the results were similar to those of mutation score. This is not surprising, as we know that mutation score subsumes instruction coverage [10]. This is an intuitive fact if one considers that a mutant can only be killed if it is covered.

#### Self-Evaluation

In Section 5.5, we observed an increase of the *Completeness* criterion (C2) when using PIT and no significant change with Reneri. It means that PIT users felt they were more complete than they initially thought while Reneri users did not change their evaluation. It can be attributed the fact that the first group (PIT) is directly exposed to the mutation score, unlike the latter (Reneri). We think that direct exposure to this metric influences the way a programmer considers his code. It was the conclusion of Delgado-Pérez et al. [35].

As for the *Assertions* criterion (C3), we observed a decrease for the PIT group and an increase for the Reneri group. First, PIT users felt that their assertions were less sufficient than they initially thought. It can be explained by the fact that mutation testing exposes them with surviving mutants, implying that there is a lack of assertions. These users have thus corrected this shortcoming. In contrast, Reneri users felt the other way. We think that it is caused by the fact that Reneri reports mostly suggested improvement with uncovered methods and not covered ones. From the tester's point of view, this would imply that existing test cases do not require any improvement, and this positively impacts self-evaluation.

#### Questionnaire

In Section 5.6, we saw that most students think that PIT leads to better programs and more skilled programmers while they think that Reneri leads to better programs a bit more than competent programmers. Once more, we can understand the difference between the two groups by considering that most Reneri users were simply told to test more methods. As test cases were written within a limited amount of time, students weren't able to attain full coverage. Consequently, when Reneri reports only contained a list of uncovered methods, students did not feel like they were learning something or improving themselves.

## 6.2 Teaching with Mutation Testing

We have explored the results from a student's point of view. We will now discuss what this means from a teacher's perspective based on the results and conclusions of related works [28, 34, 35].

First, in our case, students seem to get more out of PIT than Reneri. Classical mutation testing actually help students to improve themselves. It is a promising lead. With similar experiments, Oliveira et al. [34] and Delgado-Pérez et al. [35] got similar results. Respectively, they wrote that "the analysis states that mutation

testing can be seen as a promising testing criterion to support teaching programming foundations to novice students" and that, "as the main lesson learned, this experience shows that applying mutation testing after the manual design of test cases is an effective method to make students more aware of the need of using advanced testing techniques to increase the quality of the test suites."

Garousi et al. [28] summarised common challenges found in the literature. We highly recommend this paper to teachers who want to give a course on software testing. One of the mentioned challenges is that giving specific feedback to every student is important yet time-consuming. An automated report generation tool, such as a PIT or Reneri, can actually help in that regard. Another common challenge—the most common one—is that testing is often considered boring or tedious by students. Mutation testing can be presented as a game where the objective is to kill as most mutants as possible. This could be used to motivate students. For instance, this kind of approach has already been used with CodeDefenders [29] (see Section 2.5.1).

As we have seen in Section 6.1.2, Reneri proved to generate an insufficient amount of valuable insights. Therefore, we would rather encourage the use of classical mutation testing instead of extreme mutation testing. Yet, we think that extreme mutation testing can still be useful in some cases, especially with bigger test suites. Indeed, it could be the case that the SUT we used was too simple and that generated extreme mutants were consequently too coarsed-grained for students to miss them. Our hypothesis was that extreme mutants would be more obvious to kill than classical ones. Maybe, they were *too* obvious, making them irrelevant. One solution could be to mix the use of PIT and Reneri by using PIT first, when the test suite is small, and Reneri afterwards to go further with certain problematic tests. Stand-alone use of Reneri, however, seems ineffective.

Finally, in Section 5.6, students reported the same level of understandability regardless of whether they were exposed to PIT or Reneri reports. This indicates that code-based reports don't seem to be a problem for learning students. However, these are the only results we have on this subject, and the question needs to be investigated further.

## 6.3 Experimenting with Students

Conducting an experiment with students is challenging. In this section, we provide insights into how we addressed those challenges.

Firstly, if the experiment consists of a comparison of two groups, care must be taken in distribution of the groups. Experimenters must avoid having one

group with much more skills than the other. To cope with that, we decided to use mutation score as a criterion to separate students. This step is described in detail in Section 3.1.5. As we saw in Section 5.3, it gave us a similar distribution in both groups.

Secondly, we put our experimental design to the test with dry runs. As described in Section 3.2, three students helped us calibrating some details of the experiment. Doing this helped us to assess the actual level of students and to select an SUT according to it. In addition, it showed us that students needed help with Java itself. By not giving that help, students would have more trouble to write unit tests and results would reflect that, leading to some bias. Finally, it revealed some limitations of the tools that we had not anticipated beforehand. Therefore, we believe that it is essential to carry out dry runs before the actual experiment.

Thirdly, another challenge mentioned by Garousi et al. [28] was the difficulty for students to learn using a new tool along with learning basic concepts of software testing. To mitigate this problem, we provided students with scripts to automate the process of running PIT or Reneri with Maven. These scripts can be run directly from Visual Studio Code at the touch of a button, further simplifying the process.

A fourth challenge we faced was the difficulty of managing a large group of students (up to 26 in our case). It is necessary to be well prepared, as a large number of questions can be asked simultaneously. Dry can be useful in this regard. Another simple solution is to reduce the number of students by conducting the experiment in several rounds with smaller subgroups, at the expense of experimentation time. Finally, if a metric is to be used to separate the groups, care should be taken not to become overwhelmed by the rapid collection of all this data. One way to speed up this process is to have the students generate the metric themselves on their own machines (again, at the touch of a button) and to collect it afterwards.

## 6.4 Threats to Validity

### Conclusion validity

For a start, there is a threat to validity related to the sample size. Indeed, although one might consider 40 participants a decent number, we had to split into two groups, so we had less than 20 participants in each group. Therefore, a single student represents 5% of their group, which may lead to the belief in some cases that there is a significant difference when it is only one student. Thus, in our

analysis, we always included absolute numbers in addition to ratios.

As recommended by Papadakis et al. [11], the mutation tools in use and their mutation operators must be specified, as they can highly vary the results. See Appendix B for the exact versions and mutation operators we used.

The same authors [13] recommend dealing with *equivalent* and *redundant* mutants (see Section 2.4.2) which can pose a significant threat. We paid attention to equivalent mutants and found only one in the *GameController* class, which we did not even include in the presented results because almost no students tested it. However, we did not pay particular attention to redundant mutants. This represents a threat.

Another threat to validity lies in how we deal with missing data. Indeed, we naively assumed that the data were missing completely at random [42] for simplicity. However, this may not be the case, as some students are more likely to skip the questionnaire, for instance.

#### Internal validity

The experiment took place in our faculty's computer pool in which we required the students to use a provided computer. This ensures that everyone had a similar environment avoiding an internal threat. We also made them use *Visual Studio Code* without any extensions, which meant that they received no help from their IDE besides syntax highlighting. Nevertheless, some of them experienced software crashes, which slowed them down while we intervened.

Cooperation between students would bias their results. Therefore, we made it clear that sharing answers was not allowed at the beginning of the experiment. However, as many students were in the room, ensuring that none of them actually cooperated was challenging. Nevertheless, if some students gave each other answers, they would have a similar mutation score and thus be separated into different groups.

#### Construct validity

A common threat to validity with students is their motivation. If an exercise is not graded, students tend to take it less seriously. However, although the session was not graded, the teachers explained to the students that participating in the experiment would help them to pass the exam. Indeed, the introductory object-oriented programming course has an exam question asking students to write a unit test, while the introductory scientific research course has a question on conducting such an experiment. We think that this has encouraged motivation enough to produce significant results.

### External validity

A common threat to validity in software testing is the software under test, whether because of its complexity, size, or programming language. Previously, in Section 6.2, we saw that larger test suites with more pseudo-tested methods could lead to more insightful hints from Reneri. Thus, a system of a different size could have given a different result. However, as our participants were undergraduates and time was limited, we needed an SUT that was not too large and, at the same time, attractive enough so they would not lose interest. So a video game in the size of 2048 seemed relevant. In addition, TU Delft's *JPacman* [30] project is quite similar in size (see Section 2.5.1).

Our experiment was inspired by Delgado-Pérez et al. [35] who gave 90 minutes for the students' first testing session and 75 minutes for the second. For our experiment, we gave 60 minutes for the first part and 45 minutes for the second part (see Section 3.1.9). This is a threat to validity, as some students may have run out of time. For instance, this could impact the self-evaluation criteria: a student who has not achieved full coverage may still feel that he or she has achieved a sufficient level for the time allocated. On the contrary, some students may have felt that the time allocated was too long. Indeed, not everyone has the same attention span. Some students may have been distracted and performed worse than they could have.

Finally, all students participating in our experiment come from the same institution. However, as the results are similar to those in the literature, this does not seem to be a problem.

# Chapter 7

# Conclusion

In Chapter 1, we saw that software testing is an important field of computer science [1]. Yet, novel approaches in software testing education are needed to address identified gaps in learning [2]. We have also seen that there exist such approaches using mutation testing [29, 30].

In Chapter 2, we saw that extreme mutation testing [23] is a special case of mutation testing where entire bodies of methods are removed at once to highlight pseudo-tested methods. It is a coarse-grained technique whereas classical mutation testing is fine-grained. We wanted to know if such mutants were more obvious to kill for students, and could thus help in learning software testing fundamentals.

We designed an experiment based on previous works [34, 35] in Chapter 3 to compare the effect of two Java mutation testing tools, PIT [21] and Reneri [3], on undergraduate computer science students. The first tool uses classical mutation operators and the latter uses extreme operators. We ran the experiment with 43 second and third year students in our university while making the hypothesis that Reneri would be more effective.

However, our results in Chapter 5 indicate that students using PIT improved their test suites further than those using Reneri, thus contradicting our hypothesis. On the one hand, this reinforces the conclusions of Oliveira et al. [34] that mutation testing is effective to teach programming fundamentals and those of Delgado-Pérez et al. [35] that mutation testing helps to raise awareness of the usefulness of testing tools. On the other hand, it means that the students did not benefit as much from extreme mutation.

By looking at the results in more detail in Chapter 6, we observed that, most of the time, Reneri actually only generated method coverage hints, thus not taking advantage of what mutation testing can provide. We think this can be explained by the scale of the system under test and the allotted time of our experiment.

Therefore, our conclusion only applies to small-scale software and test suites.

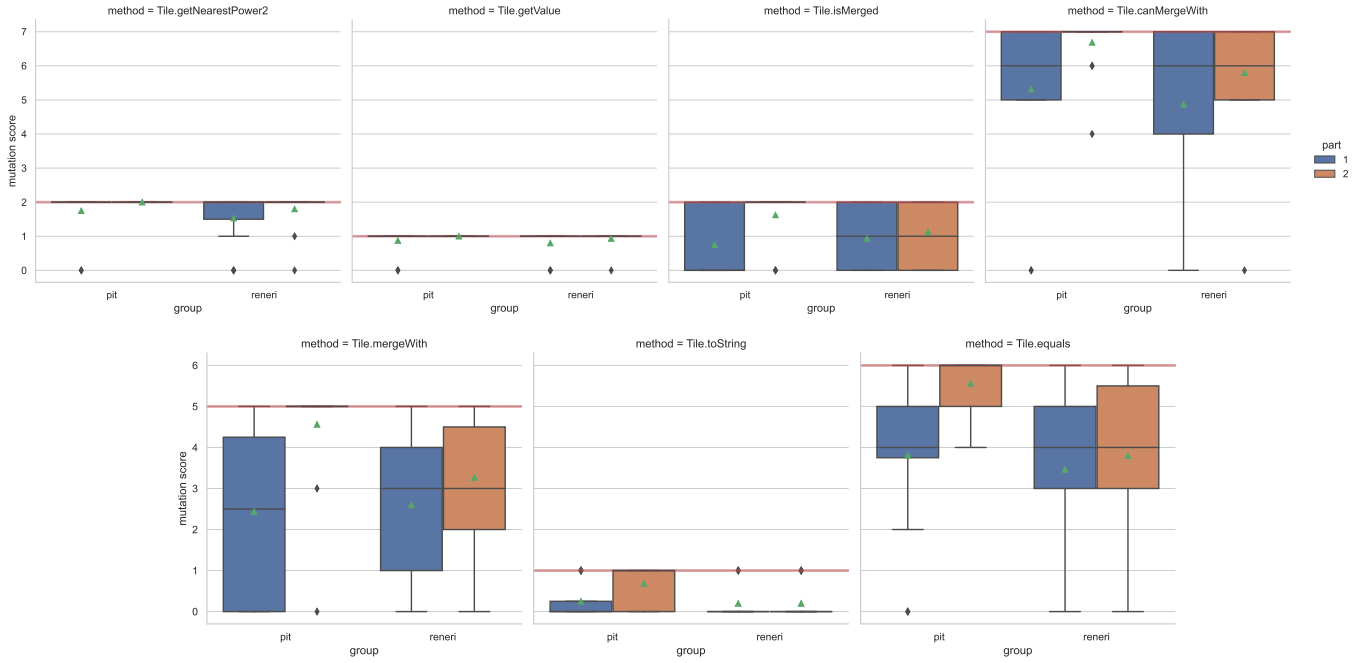
Finally, we contributed to open data by publishing all the results, students' test suites, and generated reports to *Zenodo* [36] (see Chapter 4). We summarised lessons learned concerning experimenting and the usage of mutation testing tools with students.

As for future works, we think that it would be interesting to vary parameters and see what happens with a different size SUT, an initial test suite, or more time to write test cases. Another lead that came up during the preparation of this master thesis is the comparison between code-based and text-based reports. In that matter, we could consider comparing two classical mutation testing reports: a traditional code-based one (from PIT) and a text-based one (from a tool inspired by Reneri).

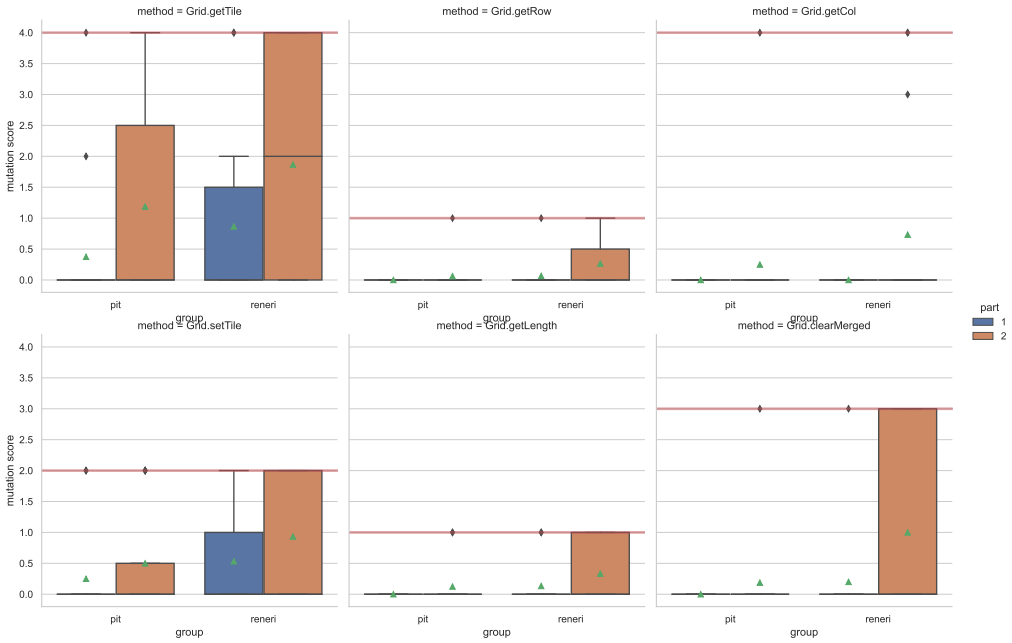
# Appendix A

# Full Data

The following pages contain the full data on mutation scores, instruction coverage, and self-assessment criteria. If you wish to find the source data (test suites written by students, Reneri reports, detailed data per student, etc.), they are available on Zenodo (see Chapter 4).




(a) Tile




(b) Grid

Figure A.1: Distribution of Killed Mutants per Method

Table A.1: Average Number of Killed Mutants

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Total | 15.8/39 (40%) | 24.4/39 (62%) | +8.6 (+22%) | 16.2/39 (41%) | 22.1/39 (56%) | +5.9 (+15%) |
| Tile | 15.2/24 (63%) | 22.1/24 (92%) | +6.9 (+28%) | 14.4/24 (60%) | 16.9/24 (70%) | +2.5 (+10%) |
| Grid | 0.6/15 (4%) | 2.3/15 (15%) | +1.7 (+11%) | 1.8/15 (12%) | 5.1/15 (34%) | +3.3 (+22%) |
| Tile.getNearestPower2 | 1.8/2 (87%) | 2.0/2 (100%) | +0.2 (+12%) | 1.5/2 (76%) | 1.8/2 (90%) | +0.3 (+13%) |
| Tile.getValue | 0.9/1 (87%) | 1.0/1 (100%) | +0.1 (+12%) | 0.8/1 (80%) | 0.9/1 (93%) | +0.1 (+13%) |
| Tile.isMerged | 0.8/2 (37%) | 1.6/2 (81%) | +0.9 (+43%) | 0.9/2 (46%) | 1.1/2 (56%) | +0.2 (+9%) |
| Tile.canMergeWith | 5.3/7 (75%) | 6.7/7 (95%) | +1.4 (+19%) | 4.9/7 (69%) | 5.8/7 (82%) | +0.9 (+13%) |
| Tile.mergeWith | 2.4/5 (48%) | 4.6/5 (91%) | +2.1 (+42%) | 2.6/5 (52%) | 3.3/5 (65%) | +0.7 (+13%) |
| Tile.toString | 0.2/1 (25%) | 0.7/1 (68%) | +0.4 (+43%) | 0.2/1 (20%) | 0.2/1 (20%) | +0.0 (+0%) |
| Tile.equals | 3.8/6 (63%) | 5.6/6 (92%) | +1.8 (+29%) | 3.5/6 (57%) | 3.8/6 (63%) | +0.3 (+5%) |
| Grid.getTile | 0.4/4 (9%) | 1.2/4 (29%) | +0.8 (+20%) | 0.9/4 (21%) | 1.9/4 (46%) | +1.0 (+25%) |
| Grid.getRow | 0.0/1 (0%) | 0.1/1 (6%) | +0.1 (+6%) | 0.1/1 (6%) | 0.3/1 (26%) | +0.2 (+20%) |
| Grid.getCol | 0.0/4 (0%) | 0.2/4 (6%) | +0.2 (+6%) | 0.0/4 (0%) | 0.7/4 (18%) | +0.7 (+18%) |
| Grid.setTile | 0.2/2 (12%) | 0.5/2 (25%) | +0.2 (+12%) | 0.5/2 (26%) | 0.9/2 (46%) | +0.4 (+20%) |
| Grid.getLength | 0.0/1 (0%) | 0.1/1 (12%) | +0.1 (+12%) | 0.1/1 (13%) | 0.3/1 (33%) | +0.2 (+20%) |
| Grid.clearMerged | 0.0/3 (0%) | 0.2/3 (6%) | +0.2 (+6%) | 0.2/3 (6%) | 1.0/3 (33%) | +0.8 (+26%) |

Table A.2: Standard Deviation Number of Killed Mutants

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Total | 6.9/39 (17%) | 6.0/39 (15%) | -0.9 (-2%) | 8.4/39 (21%) | 10.0/39 (25%) | +1.6 (+3%) |
| Tile | 6.3/24 (26%) | 3.2/24 (13%) | -3.1 (-12%) | 7.0/24 (29%) | 5.7/24 (23%) | -1.3 (-5%) |
| Grid | 1.7/15 (11%) | 4.1/15 (27%) | +2.4 (+15%) | 3.2/15 (21%) | 5.4/15 (35%) | +2.2 (+14%) |
| Tile.getNearestPower2 | 0.7/2 (34%) | 0.0/2 (0%) | -0.7 (-34%) | 0.8/2 (41%) | 0.6/2 (28%) | -0.3 (-13%) |
| Tile.getValue | 0.3/1 (34%) | 0.0/1 (0%) | -0.3 (-34%) | 0.4/1 (41%) | 0.3/1 (25%) | -0.2 (-15%) |
| Tile.isMerged | 0.9/2 (46%) | 0.8/2 (40%) | -0.1 (-6%) | 1.0/2 (48%) | 0.9/2 (45%) | -0.0 (-2%) |
| Tile.canMergeWith | 2.2/7 (31%) | 0.8/7 (11%) | -1.4 (-20%) | 2.7/7 (39%) | 1.8/7 (26%) | -0.9 (-13%) |
| Tile.mergeWith | 2.2/5 (43%) | 1.3/5 (26%) | -0.8 (-16%) | 1.9/5 (38%) | 1.5/5 (29%) | -0.4 (-8%) |
| Tile.toString | 0.4/1 (44%) | 0.5/1 (47%) | +0.0 (+3%) | 0.4/1 (41%) | 0.4/1 (41%) | +0.0 (+0%) |
| Tile.equals | 1.8/6 (30%) | 0.7/6 (12%) | -1.1 (-18%) | 2.0/6 (33%) | 2.1/6 (34%) | +0.0 (+0%) |
| Grid.getTile | 1.1/4 (27%) | 1.8/4 (43%) | +0.7 (+16%) | 1.5/4 (36%) | 1.9/4 (48%) | +0.5 (+11%) |
| Grid.getRow | 0.0/1 (0%) | 0.2/1 (25%) | +0.2 (+25%) | 0.3/1 (25%) | 0.5/1 (45%) | +0.2 (+19%) |
| Grid.getCol | 0.0/4 (0%) | 1.0/4 (25%) | +1.0 (+25%) | 0.0/4 (0%) | 1.5/4 (38%) | +1.5 (+38%) |
| Grid.setTile | 0.7/2 (34%) | 0.9/2 (44%) | +0.2 (+10%) | 0.9/2 (45%) | 1.0/2 (51%) | +0.1 (+5%) |
| Grid.getLength | 0.0/1 (0%) | 0.3/1 (34%) | +0.3 (+34%) | 0.4/1 (35%) | 0.5/1 (48%) | +0.1 (+13%) |
| Grid.clearMerged | 0.0/3 (0%) | 0.8/3 (25%) | +0.8 (+25%) | 0.8/3 (25%) | 1.5/3 (48%) | +0.7 (+22%) |

Table A.3: First Quartile Number of Killed Mutants

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Total | 12.8/39 (32%) | 21.0/39 (53%) | +8.2 (+21%) | 10.5/39 (26%) | 15.5/39 (39%) | +5.0 (+12%) |
| Tile | 12.8/24 (53%) | 21.0/24 (87%) | +8.2 (+34%) | 10.5/24 (43%) | 14.0/24 (58%) | +3.5 (+14%) |
| Grid | 0.0/15 (0%) | 0.0/15 (0%) | +0.0 (+0%) | 0.0/15 (0%) | 0.0/15 (0%) | +0.0 (+0%) |
| Tile.getNearestPower2 | 2.0/2 (100%) | 2.0/2 (100%) | +0.0 (+0%) | 1.5/2 (75%) | 2.0/2 (100%) | +0.5 (+25%) |
| Tile.getValue | 1.0/1 (100%) | 1.0/1 (100%) | +0.0 (+0%) | 1.0/1 (100%) | 1.0/1 (100%) | +0.0 (+0%) |
| Tile.isMerged | 0.0/2 (0%) | 2.0/2 (100%) | +2.0 (+100%) | 0.0/2 (0%) | 0.0/2 (0%) | +0.0 (+0%) |
| Tile.canMergeWith | 5.0/7 (71%) | 7.0/7 (100%) | +2.0 (+28%) | 4.0/7 (57%) | 5.0/7 (71%) | +1.0 (+14%) |
| Tile.mergeWith | 0.0/5 (0%) | 5.0/5 (100%) | +5.0 (+100%) | 1.0/5 (20%) | 2.0/5 (40%) | +1.0 (+20%) |
| Tile.toString | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Tile.equals | 3.8/6 (62%) | 5.0/6 (83%) | +1.2 (+20%) | 3.0/6 (50%) | 3.0/6 (50%) | +0.0 (+0%) |
| Grid.getTile | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) |
| Grid.getRow | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Grid.getCol | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) |
| Grid.setTile | 0.0/2 (0%) | 0.0/2 (0%) | +0.0 (+0%) | 0.0/2 (0%) | 0.0/2 (0%) | +0.0 (+0%) |
| Grid.getLength | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Grid.clearMerged | 0.0/3 (0%) | 0.0/3 (0%) | +0.0 (+0%) | 0.0/3 (0%) | 0.0/3 (0%) | +0.0 (+0%) |

Table A.4: Median Number of Killed Mutants

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Total | 15.0/39 (38%) | 24.0/39 (61%) | +9.0 (+23%) | 17.0/39 (43%) | 22.0/39 (56%) | +5.0 (+12%) |
| Tile | 15.0/24 (62%) | 24.0/24 (100%) | +9.0 (+37%) | 14.0/24 (58%) | 18.0/24 (75%) | +4.0 (+16%) |
| Grid | 0.0/15 (0%) | 0.0/15 (0%) | +0.0 (+0%) | 0.0/15 (0%) | 6.0/15 (40%) | +6.0 (+40%) |
| Tile.getNearestPower2 | 2.0/2 (100%) | 2.0/2 (100%) | +0.0 (+0%) | 2.0/2 (100%) | 2.0/2 (100%) | +0.0 (+0%) |
| Tile.getValue | 1.0/1 (100%) | 1.0/1 (100%) | +0.0 (+0%) | 1.0/1 (100%) | 1.0/1 (100%) | +0.0 (+0%) |
| Tile.isMerged | 0.0/2 (0%) | 2.0/2 (100%) | +2.0 (+100%) | 1.0/2 (50%) | 1.0/2 (50%) | +0.0 (+0%) |
| Tile.canMergeWith | 6.0/7 (85%) | 7.0/7 (100%) | +1.0 (+14%) | 6.0/7 (85%) | 6.0/7 (85%) | +0.0 (+0%) |
| Tile.mergeWith | 2.5/5 (50%) | 5.0/5 (100%) | +2.5 (+50%) | 3.0/5 (60%) | 3.0/5 (60%) | +0.0 (+0%) |
| Tile.toString | 0.0/1 (0%) | 1.0/1 (100%) | +1.0 (+100%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Tile.equals | 4.0/6 (66%) | 6.0/6 (100%) | +2.0 (+33%) | 4.0/6 (66%) | 4.0/6 (66%) | +0.0 (+0%) |
| Grid.getTile | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) | 0.0/4 (0%) | 2.0/4 (50%) | +2.0 (+50%) |
| Grid.getRow | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Grid.getCol | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) |
| Grid.setTile | 0.0/2 (0%) | 0.0/2 (0%) | +0.0 (+0%) | 0.0/2 (0%) | 0.0/2 (0%) | +0.0 (+0%) |
| Grid.getLength | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Grid.clearMerged | 0.0/3 (0%) | 0.0/3 (0%) | +0.0 (+0%) | 0.0/3 (0%) | 0.0/3 (0%) | +0.0 (+0%) |

Table A.5: Third Quartile Number of Killed Mutants

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Total | 20.5/39 (52%) | 27.5/39 (70%) | +7.0 (+17%) | 22.0/39 (56%) | 29.0/39 (74%) | +7.0 (+17%) |
| Tile | 19.2/24 (80%) | 24.0/24 (100%) | +4.8 (+19%) | 20.5/24 (85%) | 21.0/24 (87%) | +0.5 (+2%) |
| Grid | 0.0/15 (0%) | 3.8/15 (25%) | +3.8 (+25%) | 2.0/15 (13%) | 10.0/15 (66%) | +8.0 (+53%) |
| Tile.getNearestPower2 | 2.0/2 (100%) | 2.0/2 (100%) | +0.0 (+0%) | 2.0/2 (100%) | 2.0/2 (100%) | +0.0 (+0%) |
| Tile.getValue | 1.0/1 (100%) | 1.0/1 (100%) | +0.0 (+0%) | 1.0/1 (100%) | 1.0/1 (100%) | +0.0 (+0%) |
| Tile.isMerged | 2.0/2 (100%) | 2.0/2 (100%) | +0.0 (+0%) | 2.0/2 (100%) | 2.0/2 (100%) | +0.0 (+0%) |
| Tile.canMergeWith | 7.0/7 (100%) | 7.0/7 (100%) | +0.0 (+0%) | 7.0/7 (100%) | 7.0/7 (100%) | +0.0 (+0%) |
| Tile.mergeWith | 4.2/5 (85%) | 5.0/5 (100%) | +0.8 (+15%) | 4.0/5 (80%) | 4.5/5 (90%) | +0.5 (+10%) |
| Tile.toString | 0.2/1 (25%) | 1.0/1 (100%) | +0.8 (+75%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Tile.equals | 5.0/6 (83%) | 6.0/6 (100%) | +1.0 (+16%) | 5.0/6 (83%) | 5.5/6 (91%) | +0.5 (+8%) |
| Grid.getTile | 0.0/4 (0%) | 2.5/4 (62%) | +2.5 (+62%) | 1.5/4 (37%) | 4.0/4 (100%) | +2.5 (+62%) |
| Grid.getRow | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 0.5/1 (50%) | +0.5 (+50%) |
| Grid.getCol | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) |
| Grid.setTile | 0.0/2 (0%) | 0.5/2 (25%) | +0.5 (+25%) | 1.0/2 (50%) | 2.0/2 (100%) | +1.0 (+50%) |
| Grid.getLength | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 1.0/1 (100%) | +1.0 (+100%) |
| Grid.clearMerged | 0.0/3 (0%) | 0.0/3 (0%) | +0.0 (+0%) | 0.0/3 (0%) | 3.0/3 (100%) | +3.0 (+100%) |

Table A.6: Interquartile Range Number of Killed Mutants

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Total | 7.8/39 (19%) | 6.5/39 (16%) | -1.2 (-3%) | 11.5/39 (29%) | 13.5/39 (34%) | +2.0 (+5%) |
| Tile | 6.5/24 (27%) | 3.0/24 (12%) | -3.5 (-14%) | 10.0/24 (41%) | 7.0/24 (29%) | -3.0 (-12%) |
| Grid | 0.0/15 (0%) | 3.8/15 (25%) | +3.8 (+25%) | 2.0/15 (13%) | 10.0/15 (66%) | +8.0 (+53%) |
| Tile.getNearestPower2 | 0.0/2 (0%) | 0.0/2 (0%) | +0.0 (+0%) | 0.5/2 (25%) | 0.0/2 (0%) | -0.5 (-25%) |
| Tile.getValue | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Tile.isMerged | 2.0/2 (100%) | 0.0/2 (0%) | -2.0 (-100%) | 2.0/2 (100%) | 2.0/2 (100%) | +0.0 (+0%) |
| Tile.canMergeWith | 2.0/7 (28%) | 0.0/7 (0%) | -2.0 (-28%) | 3.0/7 (42%) | 2.0/7 (28%) | -1.0 (-14%) |
| Tile.mergeWith | 4.2/5 (85%) | 0.0/5 (0%) | -4.2 (-85%) | 3.0/5 (60%) | 2.5/5 (50%) | -0.5 (-10%) |
| Tile.toString | 0.2/1 (25%) | 1.0/1 (100%) | +0.8 (+75%) | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) |
| Tile.equals | 1.2/6 (20%) | 1.0/6 (16%) | -0.2 (-4%) | 2.0/6 (33%) | 2.5/6 (41%) | +0.5 (+8%) |
| Grid.getTile | 0.0/4 (0%) | 2.5/4 (62%) | +2.5 (+62%) | 1.5/4 (37%) | 4.0/4 (100%) | +2.5 (+62%) |
| Grid.getRow | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 0.5/1 (50%) | +0.5 (+50%) |
| Grid.getCol | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) | 0.0/4 (0%) | 0.0/4 (0%) | +0.0 (+0%) |
| Grid.setTile | 0.0/2 (0%) | 0.5/2 (25%) | +0.5 (+25%) | 1.0/2 (50%) | 2.0/2 (100%) | +1.0 (+50%) |
| Grid.getLength | 0.0/1 (0%) | 0.0/1 (0%) | +0.0 (+0%) | 0.0/1 (0%) | 1.0/1 (100%) | +1.0 (+100%) |
| Grid.clearMerged | 0.0/3 (0%) | 0.0/3 (0%) | +0.0 (+0%) | 0.0/3 (0%) | 3.0/3 (100%) | +3.0 (+100%) |

Table A.7: Average Number of Instructions Covered

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Tile | 86.5/114 (75%) | 105.4/114 (92%) | +18.9 (+16%) | 85.5/114 (75%) | 95.1/114 (83%) | +9.6 (+8%) |
| Grid | 6.7/121 (5%) | 21.9/121 (18%) | +15.2 (+12%) | 19.7/121 (16%) | 45.5/121 (37%) | +25.8 (+21%) |

Table A.8: Standard Deviation Number of Instructions Covered

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Tile | 21.3/114 (18%) | 8.9/114 (7%) | -12.4 (-10%) | 23.1/114 (20%) | 15.8/114 (13%) | -7.3 (-6%) |
| Grid | 14.7/121 (12%) | 32.9/121 (27%) | +18.2 (+15%) | 28.8/121 (23%) | 47.2/121 (39%) | +18.4 (+15%) |

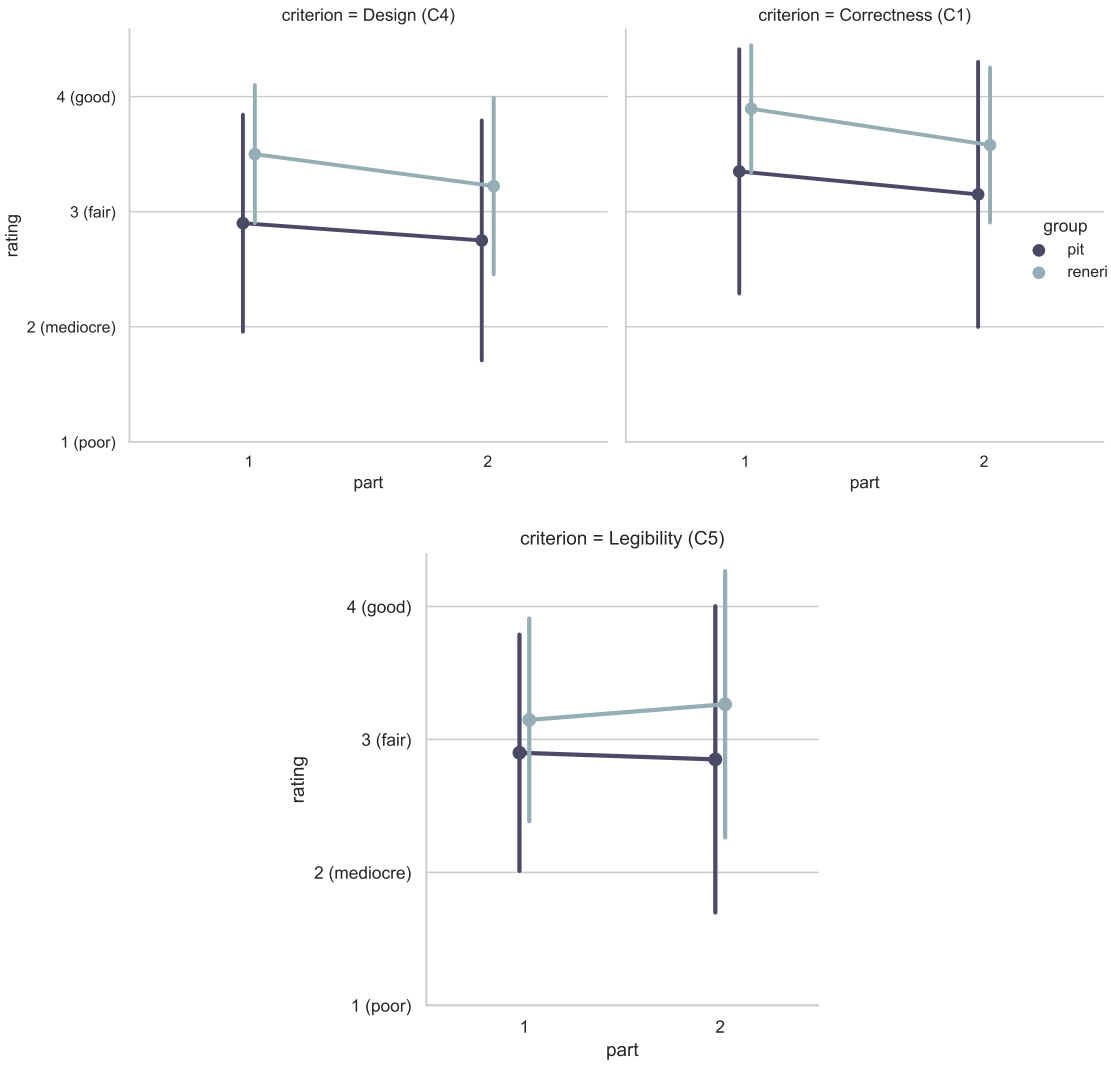



Figure A.2: Evolution of C1, C4, and C5

Table A.9: Average Rating

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Correctness (C1) | +3.35 | +3.15 | -0.20 | +3.89 | +3.58 | -0.32 |
| Completeness (C2) | +1.75 | +2.10 | +0.35 | +2.21 | +2.13 | -0.08 |
| Assertions (C3) | +2.45 | +2.25 | -0.20 | +2.53 | +2.87 | +0.34 |
| Design (C4) | +2.90 | +2.75 | -0.15 | +3.50 | +3.22 | -0.28 |
| Legibility (C5) | +2.90 | +2.85 | -0.05 | +3.15 | +3.26 | +0.12 |

Table A.10: Standard Deviation Rating

| **Group** | pit | | | reneri | | |
|---|---|---|---|---|---|---|
| **Part** | 1 | 2 | Δ | 1 | 2 | Δ |
| Correctness (C1) | +1.09 | +1.18 | +0.09 | +0.57 | +0.69 | +0.13 |
| Completeness (C2) | +1.07 | +1.12 | +0.05 | +0.92 | +0.88 | -0.04 |
| Assertions (C3) | +0.94 | +1.12 | +0.17 | +0.84 | +0.94 | +0.10 |
| Design (C4) | +0.97 | +1.07 | +0.10 | +0.62 | +0.79 | +0.17 |
| Legibility (C5) | +0.91 | +1.18 | +0.27 | +0.79 | +1.03 | +0.25 |

# Appendix B

# Versions and Mutation Operators

- Java 11.
- JUnit 4.13.2.
- JaCoCo 0.8.8.
- PIT 1.7.4.
- Descartes 1.3.2.
- Visual Studio Code 1.52.1.
- PIT default mutation operators (i.e., *Conditionals Boundary*, *Increments*, *Invert Negatives*, *Math*, *Negate Conditionals*, *Return Values*, *Void Method Calls*, *Empty returns*, *False Returns*, *True returns*, *Null returns*, *Primitive returns*).[1]
- Descartes default mutation operators (i.e., `void`, `null`, `true`, `false`, `empty`, `0`, `1`, `(byte)0`, `(byte)1`, `(short)0`, `(short)1`, `0L`, `1L`, `0.0`, `1.0`, `0.0f`, `1.0f`, `'\40'`, `'A'`, `""`, `"A"`).[2]

[1] Refer to `https://pitest.org/quickstart/mutators/`.

[2] Refer to `https://github.com/STAMP-project/pitest-descartes#specifying-operators`.

# Bibliography


[1] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2016.

[2] 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 *2018 IEEE Frontiers in Education Conference (FIE)*, pp. 1–8, 2018.

[3] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "Suggestions on test suite improvements with automatic infection and propagation analysis," *arXiv preprint arXiv:1909.04770*, 2019.

[4] A. P. Mathur, *Foundations of software testing, 2/e*. Pearson Education India, 2013.

[5] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[6] R. Osherove, *The Art of Unit Testing: with examples in C*. Simon and Schuster, 2013.

[7] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[8] A. H. Watson, D. R. Wallace, and T. J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*, vol. 500. US Department of Commerce, Technology Administration, National Institute of . . . , 1996.

[9] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[10] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, p. 597–608, IEEE Press, 2017.

[11] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," vol. 112 of *Advances in Computers*, pp. 275–378, Elsevier, 2019.

[12] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, p. 99–118, apr 1996.

[13] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), p. 354–365, Association for Computing Machinery, 2016.

[14] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," in *Advances in Computers*, vol. 112, pp. 275–378, Elsevier, 2019.

[15] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[16] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 936–946, IEEE, 2015.

[17] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *2010 Asia Pacific Software Engineering Conference*, pp. 300–309, IEEE, 2010.

[18] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[19] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, p. 5–20, jan 1992.

[20] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, 2017.

[21] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), p. 449–452, Association for Computing Machinery, 2016.

[22] L. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, 1990.

[23] R. Niedermayr, E. Juergens, and S. Wagner, "Will my tests tell me if i break this code?," in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, pp. 23–29, 2016.

[24] O. L. Vera-Pérez, M. Monperrus, and B. Baudry, "Descartes: A pitest engine to detect pseudo-tested methods: Tool demonstration," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 908–911, 2018.

[25] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "A comprehensive study of pseudo-tested methods," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1195–1225, 2019.

[26] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient java bytecode translators," in *International Conference on Generative Programming and Component Engineering*, pp. 364–376, Springer, 2003.

[27] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[28] V. Garousi, A. Rainer, P. Lauvås, and A. Arcuri, "Software-testing education: A systematic literature mapping," *Journal of Systems and Software*, vol. 165, p. 110570, 2020.

[29] G. Fraser, A. Gambi, M. Kreis, and J. M. Rojas, "Gamifying a software testing course with code defenders," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, (New York, NY, USA), p. 571–577, Association for Computing Machinery, 2019.

[30] M. Aniche, F. Hermans, and A. van Deursen, "Pragmatic software testing education," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, (New York, NY, USA), p. 414–420, Association for Computing Machinery, 2019.

[31] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 787–805, 2013.

[32] R. Ramler, T. Wetzlmaier, and C. Klammer, "An empirical study on the application of mutation testing for a safety-critical industrial software system," in *Proceedings of the Symposium on Applied Computing*, SAC '17, (New York, NY, USA), p. 1401–1408, Association for Computing Machinery, 2017.

[33] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney, "Applying mutation analysis on kernel test suites: An experience report," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 110–115, 2017.

[34] R. A. P. Oliveira, L. B. R. Oliveira, B. B. P. Cafeo, and V. H. S. Durelli, "Evaluation and assessment of effects on exploring mutation testing in programming courses," in *2015 IEEE Frontiers in Education Conference (FIE)*, pp. 1–9, 2015.

[35] P. Delgado-Pérez, I. Medina-Bulo, M. A. Álvarez-García, and K. 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)*, pp. 231–240, 2021.

[36] European Organization For Nuclear Research and OpenAIRE, "Zenodo," 2013.

[37] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (F. Loizides and B. Schmidt, eds.), pp. 87 – 90, IOS Press, 2016.

[38] T. pandas development team, "pandas-dev/pandas: Pandas," feb 2020.

[39] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference* (Stéfan van der Walt and Jarrod Millman, eds.), pp. 56 – 61, 2010.

[40] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021.

[41] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[42] J. Scheffer, "Dealing with missing data," 2002.

[43] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[44] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*, pp. 196–202, Springer, 1992.

[45] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[46] A. Vargha and H. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.