Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

UNIVERSITÉ DE NAMUR

University of Namur

RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

An Empirical Evaluation of Regular and Extreme Mutation Testing for Teaching Software Testing

Balfroid, M.; Luycx, Pierre; Vanderose, B.; Devroey, X.

Published in:
Proceedings - 2023 IEEE 16th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2023

DOI:
10.1109/icstw58534.2023.00074

Publication date:
2023

Link to publication

Citation for pulished version (HARVARD):
Balfroid, M, Luycx, P, Vanderose, B & Devroey, X 2023, An Empirical Evaluation of Regular and Extreme Mutation Testing for Teaching Software Testing. in *Proceedings - 2023 IEEE 16th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2023.* Proceedings - 2023 IEEE 16th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2023, IEEE, pp. 405-412, 2nd Software Testing Education Workshop (TestEd '23), Dublin, Ireland, 16/04/23. https://doi.org/10.1109/icstw58534.2023.00074

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 27. Apr. 2024

An Empirical Evaluation of Regular and Extreme Mutation Testing for Teaching Software Testing

Martin Balfroid
NADI, University of Namur
Namur, Belgium
martin.balfroid@unamur.be

Pierre Luycx
NADI, University of Namur
Namur, Belgium

Benoît Vanderose
NADI, University of Namur
Namur, Belgium
benoit.vanderose@unamur.be

Xavier Devroey
NADI, University of Namur
Namur, Belgium
xavier.devroey@unamur.be

***Abstract*—Teaching software testing can be challenging due to low student interest, high cognitive load, and lack of alignment with industry needs. Previous research has attempted to address these challenges by using mutation testing, which involves intentionally introducing faults into the code to measure the ability of a test suite to detect faults. Although this method has been proven effective in teaching software testing, it can sometimes be difficult for a novice to write a test to kill some mutants because they are too subtle and there are no hints. In contrast, extreme mutation testing involves more evident changes (e.g., removing a method body) that may be easier for novice testers to identify. This paper investigates extreme mutation testing as an alternative to teaching software testing by comparing it to regular mutation testing in an empirical evaluation with two undergraduate classes. Our results show that both can help teach software testing, with regular mutation testing slightly more effective, and both types of reports were considered clear by a similar number of students.**

***Index Terms*—Software Testing Education, Mutation Testing, Extreme Mutation Testing**

I. Introduction

Software testing can be challenging to teach for various reasons. These include a lack of student motivation due to the *tediousness of writing tests*, the need to *learn new tools and libraries*, and the *increased cognitive load of learning tests alongside learning programming* [1]. In this study, we focus on three Learning Challenges (LCs): lack of motivation or interest (LC1), increased cognitive load due to learning testing, programming and new tools simultaneously (LC2), and alignment with the industry needs (LC3).

Related work has addressed software testing education challenges in several ways, for instance, by relying on automated platforms [2]–[4] where students can submit their code and receive feedback on the quality of their tests. This feedback can include more than code coverage [5], for instance, non-functional aspects such as readability and understandability [6], [7]. Also, mutation analysis [8], which involves artificially introducing defects (called mutants) to assess the ability of tests to ”kill” them, i.e. to detect defects [8]. Code Defenders [9] is a game in which one team creates mutants, and the other writes tests to kill them. Mutation testing tools, such as PIT [10], have also been used in industrial settings [11], [12].

Mutation testing is effective for teaching software testing [13], [14] because it brings gamification (addressing LC1), can decrease the learning curve with the clear reports generated by mutation testing tools (addressing LC2) and is already used in industrial settings (addressing LC3). Yet mutations can often be complex for inexperienced developers to kill due to their subtle nature [15], [16] and a lack of guidance on how to write the test that would kill it.

To alleviate this issue, extreme mutation testing involves replacing an entire block of code rather than introducing a small error, which can help identify methods that have not been thoroughly tested (called pseudo-tested methods) [17]. For example, it can reveal if a boolean method has only been tested for the true case, not the false one. Thus, it may be more effective at addressing LC2 with the more apparent mutations it introduces. Vera-Pérez et al. developed a tool called Reneri [18], which combined extreme mutation with program analysis to provide suggestions for killing extreme mutants that are still alive. This tool may help simplify the interpretation of results, also addressing LC2. To our knowledge, extreme mutation testing has not been used in software testing education.

This paper compares the effectiveness of regular and extreme mutation testing in teaching software testing by conducting an empirical evaluation with 43 undergraduate students using PIT and Reneri on a Java implementation of the game 2048. The results showed that both approaches are beneficial. However, regular mutation testing is slightly more effective. Nonetheless, students found both reports to be clear.

In summary, our paper contributions are: (i) a replication of previous qualitative studies in another context, confirming previous results on mutation testing for education [13], [14]; (ii) an evaluation of the effectiveness of using extreme mutation analysis for teaching software testing; and (iii) a comparison of regular and extreme mutation operators in software testing education. Our replication package, which includes collected tests and data, is available on *Zenodo* [19].

II. Background

A. Mutation Testing

Mutation testing involves introducing defects into a program's source code to produce *mutants* [8], which are then tested against a test suite to determine its ability to detect defects. If at least one test fails, the mutant is considered *killed*, while if all tests pass, the mutant is considered *live*.

TABLE I
BASIC MUTATION OPERATORS SET [8], [20].

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

Thus, the test suite can be improved by writing a test that will fail against the mutant but not the original code.

Defects are introduced into source code using *mutation operators*, which create minor variations in the code. For example, replacing a "+" with a "−" in an arithmetic expression. The minimum standard set proposed by Offutt et al. [8], [20] is described in Table I using examples applied to Java code. The original code is presented at the top, and the mutant code is presented below.

The *mutation score* measures the effectiveness of a test suite in detecting defects. It is calculated as the number of mutants killed out of the total number of mutants and can be used to identify what to test, determine when testing is considered complete, and assess the reliability of a test suite [21]. However, it is important to note that the mutation score may be biased as not all mutations contribute equally [15]: *equivalent* mutants return the same output as the original program and are thus undetectable, and *redundant* mutants are killed alongside other mutants and thus do not contribute much to the testing process. Detecting these kinds of mutants is one of the main challenges in mutation testing [8], [15], as it is undecidable [22].

B. Reachability, Infection, Propagation and Revealability

The Reachability, Infection and Propagation (RIP) [23] model states that for a program fault to be detected, it must be *reached* during execution, *infect* the program by changing its state, and have this state change *propagate* to the output. The RIPR model [24] adds an additional requirement for a fault to be detected: it must be *revealed* through an assertion in addition to being propagated. If a fault propagates to the program state of a test case but is not revealed through an assertion, the test will not detect it.

C. Extreme Mutation Testing

A special class of operators, called *extreme mutation operators* [17], remove the body of methods or replaces it with trivial return statements (see an example in Listings 1 and 2). Extreme mutation operators have been designed to automate the detection of *pseudo-tested methods* [17]. A method is pseudo-tested when the test suite only superficially tests its side effects, which could lead to undetected failures. These methods can be identified if the test suite covers them, yet

```
public class Amount {
    private int value = 0;

    @Override
    public boolean equals(Object that) {
        if (that == null) return false;
        if (!(that instanceof Amount))
            return false;
        return this.value==((Amount)that).value;
    }
}
```

Listing 1. Original code of the method `equals`.

```
public class Amount {
    private int value = 0;

    @Override
    public boolean equals(Object that) {
        return true;
    }
}
```

Listing 2. Method `equals` after an extreme mutation.

no tests fail on extreme mutants. For instance, if the method `equals` in Listing 1 is not called with different objects by any test (i.e., only the `true` cases are tested), it is pseudo-tested and the mutant of Listing 2 will live.

Niedermayr et al. [17] applied extreme mutation testing to 14 Java open-source projects and found that, on average, 11.% (resp. 35.48%) of methods are pseudo-tested in unit (resp. system) tests. Vera-Pérez et al. developed *Descartes* [25] (a mutation engine for PIT [10], a state-of-the-art mutation testing tool for Java) to reproduce those results on a different dataset and further confirm their findings with developers.

Vera-Pérez et al. developed Reneri [18], a tool that leverages Descartes to generate reports with suggestions for improving a test suite based on identified pseudo-tested methods and the RIPR model. For example, the Reneri report for a pseudo-tested `equals` method is shown in Figure 3a. Vera-Pérez et al. [18] evaluated Reneri on 15 open-source software projects. The developers of 4 of these projects were contacted and presented with generated suggestions, which were helpful and even pointed to the exact solution in some cases. To the best of our knowledge, the Reneri approach has never been used in a software testing education context.

III. MOTIVATION

Mutation analysis is effective for teaching software testing [13], [14]. Still, it can be challenging for students and junior developers to write an appropriate test due to some mutations' nature and the lack of guidance on what test to create. Following the classification on automated feedback from Serral and Snoeck [26], the mutation score provides *summative* feedback to the students. In addition, classical mutation testing tools such as PIT provide *informative* feedback as reports with the mutation score and undetected mutants, while Reneri provides additional *suggestive* feedback through reports containing suggestions to enhance the test suite.

To the best of our knowledge, Reneri has only been evaluated with open-source developers [18]. Therefore, our research aims to determine if feedback through Reneri's *suggestive* reports based on extreme mutation and RIPR analysis can

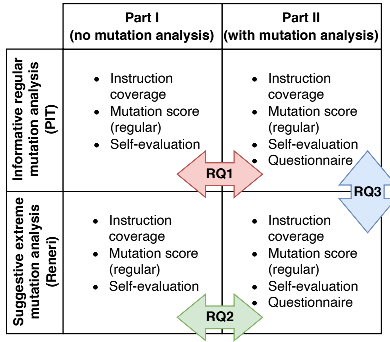

Fig. 1. RQ1 assesses the improvement in students' understanding of software testing after receiving an *informative* regular mutation analysis report, RQ2 assesses the improvement in students after receiving a *suggestive* extreme mutation analysis report, and RQ3 compares the improvements of both groups.

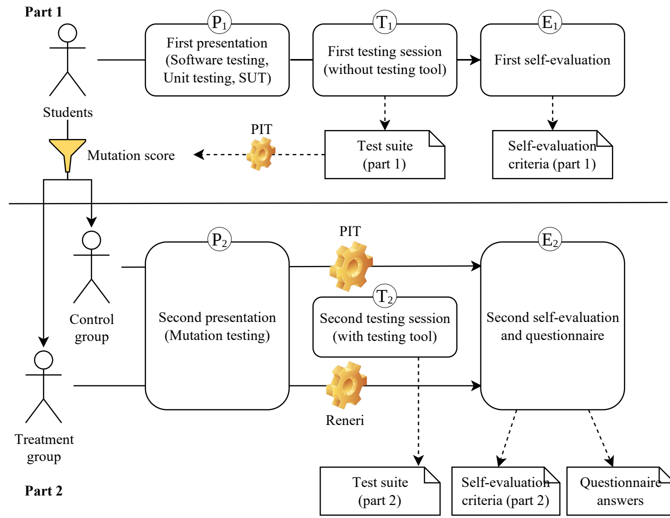

Fig. 2. Evaluation overview

also benefit undergraduate computer science students with little prior knowledge of software testing and compare it to PIT *informative* reports based on regular mutation analysis. The following research questions are outlined in Figure 1: **(RQ1)** *To what extent are informative reports effective for teaching the basics of software testing to beginner students?* Consequently, answering RQ1 requires to replicates the results of Oliveira et al. [13] and Delgado-Pérez et al. [14] using PIT in our new context, the computer science undergraduates at the University of Namur with the Java programming language. **(RQ2)** *To what extent are suggestive reports effective for teaching the basics of software testing to beginner students?* This second research question focuses on Reneri, while the third research questions compares usages of PIT and Reneri. **(RQ3)** *How does the effectiveness of suggestive reports compare to that of informative reports for teaching the basics of software testing to beginner students?*

TABLE II
EVALUATION TASKS AND DURATION

	Task	Duration
(P1)	Unit testing introduction and SUT presentation	30 minutes
(T1)	Testing session without mutation analysis	60 minutes
(E1)	First self-evaluation	5 minutes
	Break	15 minutes
(P2)	Introduction to Mutation testing	10 minutes
(T2)	Testing session with mutation analysis	45 minutes
(E2)	Second self-evaluation	5 minutes
(E2)	Questionnaire	10 minutes
Total		**180 minutes**

IV. EVALUATION SETUP

Our empirical evaluation involved 43 computer science undergraduates from the University of Namur divided into two classes. The first class included 26 second-year students, while the second class included 17 third-year students. Inspired by both the work of Delgado-Pérez et al. [14] and Oliveira et al. [13], the evaluation was divided into two phases. In the first phase, students wrote unit tests for a Java game. In the second phase, students were divided into two groups based on their mutation scores and received either a PIT (v1.7.4) or Reneri (v1.0-EXPERIMENTS) report to improve their test suite. We measured the quality of the tests using instruction coverage and mutation scores. We also assessed the impact of each reporting approach on learning with a self-evaluation and a questionnaire. Figure 2 presents an overview of the different steps and the duration of each step can be found in Table II.

Before the first test writing session (P1), we introduced the students to unit testing in Java. This introduction explained the importance of automated testing, what a unit test is, and how to write one. We also presented the System Under Test (SUT), an adapted version of the 2048 game written in Java[1], described in Section IV-A. During the first test writing session (T1), students were asked to manually write unit tests on the given SUT without any tool. At the end of this session (E1), we asked the students to fill in a self-evaluation form.

After the first session, there was a break during which the students were split into two groups based on their mutation scores. To preserve a similar distribution of skills, we ranked the students according to the mutation scores of their initial test suite and distributed similar students into the two groups.[2] The control group used PIT and the treatment group used Reneri.

Before the second test writing session (P2), we introduced the students to the main concepts of mutation testing. During this session (T2), students were asked to use the report provided by the mutation testing tool they were assigned to improve their test suite by adding or editing test cases. They were able to regenerate the report at any time to update it. At the end of the second session (E2), students completed a

[1] The original implementation can be found on *Rosetta Code*: https://www.rosettacode.org/wiki/2048\#Java.

[2] We used PIT with the default mutation operators: *Conditionals Boundary*, *Increments*, *Invert Negatives*, *Math*, *Negate Conditionals*, *Return Values*, *Void Method Calls*, *Empty returns*, *False Returns*, *True returns*, *Null returns*, *Primitive returns*.

TABLE III
SELF-EVALUATION CRITERIA [14].

C1. Correctness	The tests do not cause errors when executed on the original program.
C2. Completeness	The tests represent a sufficiently large portion of the different use scenarios of the program.
C3. Assertions	There are enough assertions in the tests to cover all the state changes that occur during execution.
C4. Design	Each test is designed to validate a single capability and there are no duplicate or overlapping tests.
C5. Legibility	The programming style is clear and makes it possible to understand the purpose of the tests and their assertions.

TABLE IV
QUESTIONNAIRE AND DISTRIBUTION OF THE ANSWERS.

	PIT	Reneri	Total
Q1. What was your knowledge of Java before this experiment?			
No knowledge	1 (5%)	0 (0%)	1 (2%)
Basic knowledge	13 (65%)	14 (67%)	27 (66%)
Intermediate knowledge	6 (30%)	5 (24%)	11 (27%)
Advanced knowledge	0 (0%)	2 (10%)	2 (5%)
Q2. What was your knowledge of software testing before this experiment?			
No knowledge	4 (20%)	4 (19%)	8 (20%)
Basic knowledge	12 (60%)	12 (57%)	24 (59%)
Intermediate knowledge	4 (20%)	4 (19%)	8 (20%)
Advanced knowledge	0 (0%)	1 (5%)	1 (2%)
Q3. Do you think it is interesting to present the concepts of mutation testing together with the basics of programming?			
Yes	8 (40%)	9 (43%)	17 (41%)
Yes but superficially	10 (50%)	10 (48%)	20 (49%)
No	2 (10%)	2 (10%)	4 (10%)
Q4. What could be the consequences of using mutation testing by novice programmers?			
Better programs	7 (35%)	3 (14%)	10 (24%)
More competent programmers	4 (20%)	2 (10%)	6 (15%)
Better programs and more competent programmers	9 (45%)	15 (71%)	24 (59%)
Neither	0 (0%)	1 (5%)	1 (2%)
Q5. Do you consider regular testing tools (JUnit) to be useful for teaching programming fundamentals?			
Yes	11 (55%)	10 (48%)	21 (51%)
Yes, but only with basic functionality	8 (40%)	9 (43%)	17 (41%)
No	1 (5%)	2 (10%)	3 (7%)
Q6. Do you consider mutation testing tools useful for teaching the fundamentals of programming?			
Yes	8 (40%)	7 (33%)	15 (37%)
Yes, but only with basic functionality	9 (45%)	12 (57%)	21 (51%)
No	3 (15%)	2 (10%)	5 (12%)
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	6 (30%)	8 (38%)	14 (34%)
Insufficiently presented	13 (65%)	13 (62%)	26 (63%)
Not presented	1 (5%)	0 (0%)	1 (2%)
Q8. Do you think using software testing tools for learning purposes could be useful for creating good programming habits?			
Yes	20 (100%)	21 (100%)	41 (100%)
No	0 (0%)	0 (0%)	0 (0%)
Q9. Do you think creating test cases through mutation testing is useful for improving the learning ability of novice programmers?			
Yes	18 (90%)	21 (100%)	39 (95%)
No	2 (10%)	0 (0%)	2 (5%)
Q10. How did you find creating tests manually without the help of a tool?			
Easy in general	5 (25%)	8 (38%)	13 (32%)
Difficult, especially with regard to the completeness of my tests (sufficient code coverage)	11 (55%)	9 (43%)	20 (49%)
Difficult, especially to follow a logical order in the design of test cases	4 (20%)	4 (19%)	8 (20%)
Q11. What is your perception of software testing after applying mutation testing to your tests?			
It has changed the way I design tests	4 (20%)	2 (10%)	6 (15%)
This allowed me to discover parts of the code that were not sufficiently tested	15 (75%)	15 (71%)	30 (73%)
The mutants do not seem to me to be particularly useful for improving the quality of my tests	1 (5%)	4 (19%)	5 (12%)
Q12. The reports generated by the tool used in the second session:			
Were sufficiently understandable	19 (95%)	20 (95%)	39 (95%)
Lacked comprehensibility but were still usable	1 (5%)	1 (5%)	2 (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	13 (65%)	13 (62%)	26 (63%)
You have overestimated yourself	5 (25%)	6 (29%)	11 (27%)
You have undervalued yourself	2 (10%)	2 (10%)	4 (10%)
Q14. From a practical point of view, mutation testing:			
Is very useful	15 (75%)	17 (81%)	32 (78%)
Is very useful but not comfortable to use	4 (20%)	4 (19%)	8 (20%)
Does not compensate for the effort required to use it	1 (5%)	0 (0%)	1 (2%)

second self-evaluation form and a questionnaire to assess the impact of each mutation testing approach on learning.

A. System Under Test

The System Under Test (SUT) is an adapted version of the popular game 2048, in which players combine numbered tiles on a 4x4 grid by moving them up, down, left, or right. When two tiles with the same value are merged, their values are added, and the previous tile becomes empty. A tile can only be merged once per move, and the game is won when a tile reaches 2048. The game is lost if no more tiles can be moved. We chose to use 2048 as the SUT because (i) it is well-known and easy to understand, and (ii) it can increase students' motivation, addressing LC1. To make the tests repeatable and simplify the testing, we modified the rules of the original 2048 game to eliminate its randomness (i.e., new tiles appear at pre-defined moments instead of randomly). The modified source code is available in our replication package [19].

For the evaluation, students were asked to write tests for three classes: (i) **Tile** (64 lines of code, 6 non-accessor methods) represents a tile by its value and whether it has already been merged for the current move; (ii) **Grid** (41 lines of code, 10 non-accessor methods) represents a two-dimensional grid of tiles; (iii) **GameController** (150 lines of code, 13 non-accessor methods) controls the grid, game state, score, high score and number of tiles added.

B. Data Collection

We collected data at each step P_i, T_i, E_i of Figure 1. The following subsections discuss the details of the data collected.

1) Mutation score and instruction coverage: We used the mutation score and instruction coverage to analyze the quality of students' final test suites and their evolution. These metrics are computed using PIT's (v1.7.4) [10] default operators at the end of the first (T1) and second (T2) testing sessions.

2) Self-Evaluation: We asked the students to rate their tests using a scale of 1-4 (from poor to good) after each test writing session (E1 and E2). These ratings, described in Table III [14], help understand how students' perceptions of their work change based on the tool used.

3) Questionnaire: In the second evaluation phase (E2), we provided the students with a questionnaire (Table IV) adapted from Delgado-Perez et al. [14] and Oliveira et al. [13] with the addition of two questions: one on the Java skills of the students to assess their knowledge and one on the understandability of the reports generated by Reneri or PIT. The questionnaire was initially written in French. The English version of the questionnaire can be found in Table IV.

C. Reneri hints

We have converted the standard text-only reports produced by Reneri into an HTML report written in French to facilitate their understanding using a Python script. An example is shown in Figure 3 (both figures feature the same hint). We designed the Python script to (i) directly generates an

```
The body of the method Tile.equals(java.lang.Object) was replaced by return
true; yet, the test Test2048.testTileEquals has not failed.
Running the tests with the altered or the original method causes no
observable difference in the program state.
Thus, one solution might be to create a variant of the tests listed above in
which the method returns a different value.
```

(a) Original Report

← 1/34 →

Tile.equals(java.lang.Object)

The body of the method `Tile.equals(java.lang.Object)` was replaced by `return true;` yet, the test `Test2048.testTileEquals` has not failed.

Running the tests with the altered or the original method causes no observable difference in the program state.

Thus, one solution might be to create a variant of the tests listed above in which the method returns a different value.

(b) Adapted Report

Fig. 3. Example of a Reneri Report

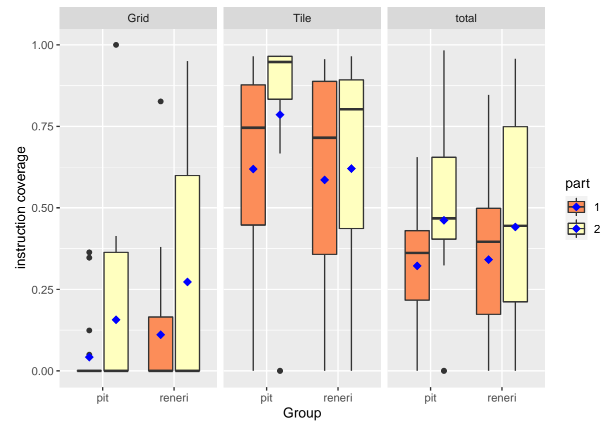

Fig. 4. Evolution of the instruction coverage ratio for each class before and after the introduction of the tool. Means are represented as blue diamonds.

HTML report instead of a text file; (ii) includes additional information about **uncovered methods** to avoid empty reports when the students did not write any valid test; (iii) **translates** it into the French.

D. Data Analysis

We use two non-parametric rank tests with a *p-value* threshold of 0.05 to analyze our data, depending on whether the observations are independent. For independent observations (RQ3 in Figure 1), we use the Mann-Whitney U test, denoted U. For dependent observations (RQ1 and RQ2 in Figure 1), we use the Wilcoxon signed-rank test, denoted W.

We also rely on Vargha-Delaney's $\hat{A}_{12}$ [27] non-parametric effect size measure. It compares the performance of two techniques by calculating the probability that a random sample from one group outperforms a random sample from the other group. Values of $\hat{A}_{12}$ above 0.5 indicate that the first group is more likely to outperform the second, while values below 0.5 indicate the opposite. We use standard thresholds to interpret the magnitude of the effect size [27]: $\hat{A}_{12} > 0.56$ 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*.

V. Evaluation Results

The data collected and answers to research questions are described below. The replication package with the artefacts and collected data is available on Zenodo [19].

A. Participants

The evaluation was conducted at the computer science faculty of the University of Namur in Belgium on students taking the "Object Oriented Design and Programming" course for the first run. This course teaches Java programming and testing to second-year undergraduate computer science and business engineering students. Third-year computer science students taking the "Introduction to Scientific Research" seminar, which introduces students to computer science research through practical exercises, were also included in the evaluation for a second run. These students have more programming experience but are still juniors in software testing.

We ran the protocol with a total of 43 undergraduates, 26 second-year and 17 third-year students, divided into two groups: 21 in the control group and 22 in the treatment group. The first group received informative reports based on a regular mutation analysis generated by PIT and the second group received suggestive reports based on an extreme mutation analysis generated by Reneri.

Based on responses to questions Q1 and Q2, 28 (68%) of the students had no or only basic knowledge of Java, while 13 (32%) had intermediate or advanced knowledge. Additionally, 32 (79%) of the students had no or only basic knowledge of software testing, while 9 (22%) had intermediate or advanced knowledge. In response to question Q7, 27 (64%) students felt that software testing was not sufficiently introduced in their studies. These results may be due to the fact that the students are still undergraduates and have not yet received focused instruction on software testing in their curriculum.

We excluded missing data from our analysis following a standard approach [28]. As a result, we had groups of 16 (control) and 15 (treatment) students for mutation scores, coverage, and hint types, and groups of 20 (control) and 19 (treatment) students for self-evaluation criteria (reducing to 18 for C4 and 17 for C5 in the second group). This was because some students had submitted a failing test suite with blank values in the datasets, while others had not completed the self-evaluation for one or more criteria.

For comparison, in our experiment, we had 21 students using a regular mutation testing tool, while Delgado-Pérez et al. [14] had 20 students and Oliveira et al. [13] had 28 students using such a tool.

B. Impact of Informative Reports (RQ1)

1) Instruction Coverage and Mutation Score: According to Figures 4 and 5, the instruction coverage and mutation score of the tests written by students improved significantly after

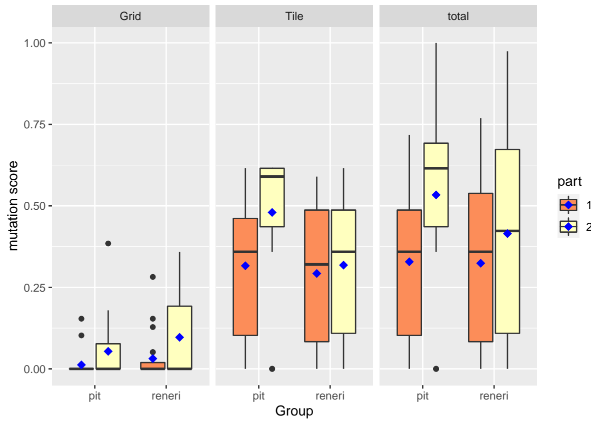

Fig. 5. Evolution of the distribution of mutation scores at the end of the first and second part for each group. Means are represented as blue diamonds.

TABLE V
AVERAGE RATING FROM 1 (POOR) TO 4 (GOOD) OF THE SELF-EVALUATION CRITERIA AFTER THE FIRST (1) AND THE SECOND (2) PHASES.

Group		pit			reneri		
Part		1	2	Δ	1	2	Δ
Correctness (C1)	μ	3.35	3.15	-0.20	3.89	3.58	-0.32
	σ	1.09	1.18	0.09	0.57	0.69	0.13
Completeness (C2)	μ	1.75	2.10	+0.35	2.21	2.13	-0.08
	σ	1.07	1.12	0.05	0.92	0.88	0.04
Assertions (C3)	μ	2.45	2.25	-0.20	2.53	2.87	+0.34
	σ	0.94	1.12	0.17	0.84	0.94	0.10
Design (C4)	μ	2.90	2.75	-0.15	3.50	3.22	-0.28
	σ	0.97	1.07	0.10	0.62	0.79	0.17
Legibility (C5)	μ	2.90	2.85	-0.05	3.15	3.26	+0.12
	σ	0.91	1.18	0.27	0.79	1.03	0.25

using PIT. The average instruction coverage increased from $32.2\%; (\sigma = 19.5)$ to $46.2\%; (\sigma = 24.1)$, with a statistically significant difference ($W = 16.00$, $p = .001$) and a large effect size ($\hat{A}_{12} = 0.73$). The average mutation score increased from $32.8\%; (\sigma = 22.4)$ to $53.4\%; (\sigma = 26.5)$, with a statistically significant difference ($W = 15.5$, $p < .001$) and a large effect size ($\hat{A}_{12} = 0.76$). These results, consistent with those of Oliveira et al. [13], suggest that using PIT helps students learn software testing and write effective tests.

2) Auto-Evaluation: According to Table V, the control group's average ratings for the criteria listed in Table III changed as follows: Completeness (C2) increased from 1.75 to 2.1, and Design (C4) decreased from 2.9 to 2.75. These trends differ from those observed by Delgado-Pérez et al. [14]. The increase in Completeness ratings suggests that students believe their tests represent a larger portion of the different use scenarios than initially thought, while the decrease in Design ratings suggests that students believe their tests are not as well designed to verify functionalities as they initially thought.

3) Questionnaire: According to the answers in Table IV, students see the advantages of mutation testing for beginners (Q3-Q6) and find that tools providing mutation analysis help them learn programming (Q8, Q9) and write tests (Q10, Q11). Similar to the results reported by Delgado-Pérez et al. [14], most students (62%, Q10) found it difficult to design test cases manually, and 71% (Q11) realized that some areas of code were not properly covered thanks to mutation analysis.

4) Summary: Informative reports based on a regular mutation analysis improve students' unit testing skills, in line with the findings of Oliveira et al. [13] and Delgado-Pérez et al. [14]. Students find weaknesses in the assertions of the tests they wrote before using PIT and report learning from the tool and feeling that their test suite is more complete than they initially thought. These results are similar to those observed by Delgado-Pérez et al. [14], who also found an improvement in assertions but a decrease in reported completeness.

C. Impact of Suggestive Reports (RQ2)

1) Instruction Coverage and Mutation Score: The Reneri group shows an increase in average instructions covered (from $34.1\%; (\sigma = 24.0)$ to $44.1\%; (\sigma = 32)$, $W = 24.00$, $p = .020$, $\hat{A}_{12} = 0.58$) and average mutation score (from $32.4\%; (\sigma = 25.4)$ to $41.5; (\sigma = 32.1)$, $W = 28.50$, $p = .020$, $\hat{A}_{12} = 0.66$) after using extreme mutation testing and receiving suggestive reports. These changes were statistically significant and had a small to medium effect size, suggesting that Reneri helped novice programmers write more effective unit tests.

2) Auto-Evaluation: According to the right part of Table V, students experienced a decrease in the ratings for Correctness (C1) and Design (C4) after the second test writing phase, with a delta of -0.32 and -0.28, respectively. This may suggest that some students initially overestimated the quality of their tests, as confirmed by 6 out of 21 students (29%) who indicated that they overestimated themselves after the first test writing session (Q13). Completeness (C2) was not impacted by the use of Reneri ($\delta = -0.08$). However, ratings for Assertions (C3) and Legibility (C5) increased with a delta of $+0.34$ and $+0.12$, respectively, suggesting that students became more confident in their assertions and the purpose of their tests after using Reneri.

3) Questionnaire: According to Table IV, most students found Reneri useful and understandable, yet one student had difficulty understanding the tool (Q12).

4) Summary: Extreme mutation analysis has a medium impact on students' unit testing skills, as indicated by the changes in instruction coverage and mutation score. The use of Reneri did not affect students' self-evaluation of the completeness of their test suites, but it did increase their confidence in the assertions written before using the tool.

D. Informative vs Suggestive Reports (RQ3)

1) Instruction Coverage and Mutation Score: According to RQ1 and RQ2, Figure 4 shows that students using PIT had an average increase of $14\%; (\sigma = 4.6)$ in instruction coverage between the first and second testing sessions, while students using Reneri had an average increase of $10\%; (\sigma = 8)$. The difference in individual increases was not statistically significant ($U = 263.50$, $p = .434$, $\hat{A}_{12} = 0.43$). Figure 5 shows that students using PIT increased their mutation

score by an average of $20.6\%; (\sigma = 4.1)$, while students using Reneri increased their mutation score by an average of $9.1\%; (\sigma = 6.7)$. The difference in individual increases was also not statistically significant ($U = 304.5$, $p = .075$, $\hat{A}_{12} = 0.34$). These results suggest that both PIT and Reneri help students achieve higher coverage and mutation scores, and there is no significant difference between the two in terms of individual increases.

2) Auto-Evaluation: Comparing the trends of the different criteria in Table V, we see that the perception of Correctness (C1) and Design (C4) follows the same direction for both groups, while Completeness (C2), Assertions (C3), and Legibility (C5) show opposite trends. This suggests that students have more confidence in their tests when receiving suggestive reports from Reneri than informative reports from PIT.

3) Questionnaire: A total of 21 students (51%) think that using testing tools such as JUnit helps to teach basic programming concepts (Q5), while 15 students (37%) felt the same way about mutation testing tools such as PIT or Reneri (Q6). However, 17 (41%) students think that using testing tools with basic functionality is helpful (Q5), and 21 (51%) find mutation testing tools with basic functionality helpful (Q6). Thus, a similar proportion of students think that using regular or mutation testing tools, at least partially, is helpful for learning. However, they mostly feel that mutation testing tools should be kept to basic functionality. This might be due to the increased cognitive load of such tools for novice programmers [1] (LC2). All students agreed that using testing tools can create good programming habits (Q8), and almost all students (95%) think designing test cases through mutation testing helps improve the learning capacity of novice programmers (Q9). 32 students (78%) think that mutation testing is beneficial, while 8 students (20%) find it useful but uncomfortable to use (Q14). In both groups, 39 students (95%) found the reports sufficiently understandable (Q12).

4) Summary: PIT and Reneri both had a positive impact on instruction coverage and mutation score of the final test suites produced by the students. According to the questionnaire, a slightly smaller number of students found Reneri useful for learning compared to PIT. However, the same number of students in both groups found the reports clear, indicating no difference between PIT and Reneri reports in terms of understandability.

VI. Discussion

A. Instruction Coverage and Mutation Score

From the analysis of RQ3 and Figures 4 and 5, there was no significant difference between the test suites written by the two groups in terms of increase in instruction coverage and mutation score after the second test writing phase. Students receiving informative reports based on a regular mutation analysis (PIT) achieved a slightly better final mutation score. One possible explanation for this is that regular mutation testing with PIT relies on finer-grained mutants, which can be trickier to kill. In contrast, extreme mutation testing with Reneri relies on coarse-grained mutants, which are easier to kill. This could lead students using Reneri to spend less time writing tests per method and cover more methods with weaker tests. However, when comparing the increases of the individual test suites, the difference was not statistically significant.

B. Informative and Suggestive Analysis Reports

On the one hand, PIT's informative reports provide a regular mutation analysis highlighting the mutants that survived and the mutation score for the whole SUT and each class. Based on this information, it is up to the student to decide how to increase their mutation score. This can be done by either writing more test cases targeting the surviving mutants in already-covered methods or targeting uncovered methods.

On the other hand, Reneri's suggestive reports present an analysis of which methods have been pseudo-tested and which have not yet been. It is up to the student to decide whether to write more test cases for the pseudo-tested methods (even if this does not include all the remaining regular mutants) or to write test cases to cover a wide range of methods first. Thus, following Reneri's hints might lead the student to prioritize covering methods before improving test assertions.

Students exposed to informative feedback (PIT) had an increase in their perception of Completeness (C2), while there was no significant change for those not exposed to it (Reneri). This may be due to PIT users having direct access to their mutation score, which was unavailable to Reneri users. According to Delgado-Pérez et al. [14], this direct access to the mutation score can affect how students view their code.

The PIT group perceived a decrease in the sufficiency of their assertions (C3), while the Reneri group perceived an increase. This may be because PIT users were directly exposed to the surviving mutants, which indicated a lack of assertions and induced PIT users to kill the mutants in the second step. On the contrary, Reneri users may have perceived an increase because it only provides specific hints without a global view of test suite quality and because its extreme mutation analysis cannot identify subtle faults. As a result, Reneri users may feel that their test cases do not need as many improvements, leading to a more optimistic self-evaluation.

Developing and evaluating alternative presentations to provide feedback to students about the overall quality of their test suite is part of our future work.

C. Threats to Validity

1) External validity: Our experiment was conducted on a sample of computer science students from a single institution, and only one SUT written in Java, the game 2048, was involved in the evaluation. Therefore, the results may not apply to students from another institution or major. They may also not apply to other SUTs written in Java or another language. Nevertheless, the results for regular mutation analysis align with previous findings in the literature [13], [14].

2) Internal validity: We controlled for potential biases by providing students with uniform development environments (desktop machines with *Visual Studio Code* without any extensions except syntax highlighting) and reminding them that

participation in the evaluation would not affect their grades. Although some students may have cooperated during the first stage, this is not a major concern because they would have been placed in different groups, since students with similar mutation scores are likely separated into different groups. Another reason for separating the groups was to mitigate the potential impact of students' prior knowledge on their performance. Finally, to address students' motivation learning challenge (LC2) [1], we emphasized to students that participating in this experiment would be beneficial in preparing for the exam, which includes questions on unit testing for second-years and empirical evaluations for third-years.

3) Construct validity: One potential issue with the experiment is using self-evaluation as a measure. Self-evaluation involves the participants assessing their performance, which can be subjective and may not accurately reflect their actual ability or learning. Additionally, junior students may have difficulty understanding and distinguishing the criteria used in the self-evaluation, which could affect the reliability of the data. To mitigate this threat, we looked at the deltas measured between the first and second parts of the evaluations. Regarding equivalent and redundant mutants, we followed standard practices [15] and excluded equivalent mutants from our analysis. However, we did not analyse redundant mutants.

4) Conclusion validity: We split our initial set of 43 participants in two groups and removed incomplete data, resulting in a sample of 16 (resp. 15) participants for the PIT (resp. Reneri) approach. This sample size is similar to that of previous studies [13], [14]. We accounted for the potential impact of a small sample size by including absolute numbers and ratios in our analysis. We assumed that missing data was missing completely at random [28], although there might be a bias towards certain student profiles producing more missing data.

VII. Conclusion

This study compared the use of suggestive reports based on an extreme analysis with Reneri to informative reports based on a regular mutation analysis with PIT for teaching software testing in an educational setting. The results showed that both methods were effective in teaching software testing, with regular mutation testing slightly more effective. The difficulty of handling the tools and the clarity of the reports were similar for both methods. In future work, the combination of multiple reports and alternative reports can be explored, depending on the test suite and desired feedback information level.

Acknowledgment

This research was partially supported by the ARIAC project (No. 2010235), funded by the Service Public de Wallonie (SPW Recherche).

References

[1] V. Garousi, A. Rainer, P. Lauvås Jr, and A. Arcuri, "Software-testing education: A systematic literature mapping," *JSS*, vol. 165, 2020.

[2] J. M. Rojas and G. Fraser, "Code defenders: A mutation testing game," in *MUTATION '16*, pp. 162–167, IEEE, 2016.

[3] S. Elbaum, S. Person, J. Dokulil, and M. Jorde, "Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable," in *ICSE '07*, pp. 688–697, IEEE, May 2007.

[4] N. Silvis-Cividjian, R. Limburg, N. Althuisius, E. Apostolov, V. Bonev, R. Jansma, G. Visser, and M. Went, "VU-BugZoo: A Persuasive Platform for Teaching Software Testing," in *ITiCSE '20*, pp. 553–553, ACM, June 2020.

[5] H. Hemmati, "How Effective Are Code Coverage Criteria?," in *QRS '15*, pp. 151–156, IEEE, Aug. 2015.

[6] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *ESEC/FSE '15*, pp. 107–118, 2015.

[7] D. Roy, M. Ma, S. Panichella, and D. Gonzalez, "DeepTC-Enhancer : Improving the Readability of Automatically Generated Tests," in *ASE '20*, pp. 339–350, ACM, 2020.

[8] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*, vol. 112, pp. 275–378, Elsevier, 2019.

[9] G. Fraser, A. Gambi, M. Kreis, and J. M. Rojas, "Gamifying a software testing course with code defenders," in *SIGCSE '19*, pp. 571–577, 2019.

[10] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *ISSTA '16*, pp. 449–452, 2016.

[11] G. Petrović and M. Ivanković, "State of mutation testing at google," in *ICSE-SEIP '18*, pp. 163–171, 2018.

[12] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, "What it would take to use mutation testing in industry—a study at facebook," in *ICSE-SEIP '21*, pp. 268–277, IEEE, 2021.

[13] R. A. Oliveira, L. B. Oliveira, B. B. Cafeo, and V. H. Durelli, "Evaluation and assessment of effects on exploring mutation testing in programming courses," in *FIE '15*, pp. 1–9, IEEE, 2015.

[14] P. Delgado-Pérez, I. Medina-Bulo, M. Á. Álvarez-García, and K. J. Valle-Gómez, "Mutation testing and self/peer assessment: analyzing their effect on students in a software testing course," in *ICSE-SEET '21*, pp. 231–240, IEEE, 2021.

[15] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *ISSTA '16*, pp. 354–365, 2016.

[16] B. C. José Miguel Rojas, Thomas White and G. Fraser, "Code Defenders: Crowdsourcing effective tests and subtle mutants with a mutation testing game," in *ICSE '17*, pp. 677–688, IEEE, 2017.

[17] R. Niedermayr, E. Juergens, and S. Wagner, "Will my tests tell me if i break this code?," in *CSED '16*, pp. 23–29, IEEE, 2016.

[18] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "Suggestions on test suite improvements with automatic infection and propagation analysis," *arXiv e-prints*, pp. arXiv–1909, 2019.

[19] M. Balfroid and P. Luycx, "balfroim/muted: 1.1." https://doi.org/10.5281/zenodo.6645253, July 2022.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *TOSEM*, vol. 5, no. 2, pp. 99–118, 1996.

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

[22] 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.

[23] L. J. Morell, "A theory of fault-based testing," *TSE*, vol. 16, no. 8, pp. 844–857, 1990.

[24] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *TSE*, vol. 43, no. 04, pp. 372–395, 2017.

[25] O. L. Vera-Pérez, M. Monperrus, and B. Baudry, "Descartes: A pitest engine to detect pseudo-tested methods: Tool demonstration," in *ASE '18*, pp. 908–911, 2018.

[26] E. Serral and M. Snoeck, "Conceptual Framework for Feedback Automation in SLEs," in *Smart Education and e-Learning 2016* (V. L. Uskov, R. J. Howlett, and L. C. Jain, eds.), vol. 59, pp. 97–107, Springer, 2016.

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

[28] J. Scheffer, "Dealing with missing data," *Research Letters in the Information and Mathematical Sciences*, vol. 3, no. 1, p. 153 – 160, 2002.