Article

Not peer-reviewed version

Moving Towards a Mutant-Based Testing Tool for Verifying Behavior Maintenance in Test Code Refactorings

Tiago Samuel Rodrigues Teixeira , Fábio Fagundes Silveira * , Eduardo Martins Guerra

Posted Date: 16 October 2023

doi: 10.20944/preprints202310.0969.v1

Keywords: software engineering; test code refactoring; test smells; mutation testing

Article

Moving towards a Mutant-Based Testing Tool for Verifying Behavior Maintenance in Test Code Refactorings

Tiago Samuel Rodrigues Teixeira [1,†], Fábio Fagundes Silveira [2,†*] and Eduardo Martins Guerra [3,†]

[1] Instituto de Pesquisas Tecnológicas – IPT, São Paulo, Brazil; tiagosamfito@gmail.com
[2] Federal University of São Paulo – UNIFESP, São José dos Campos, Brazil; fsilveira@unifesp.br
[3] Free University of Bozen-Bolzano – UNIBZ, Bozen-Bolzano, Italy; guerraem@gmail.com
* Correspondence: fsilveira@unifesp.br
† These authors contributed equally to this work.

Abstract: Evaluating mutation testing behavior can help decide whether refactoring successfully maintains the expected initial test results. Moreover, manually performing this analytical work is both time-consuming and prone to errors. This paper extends an approach to assess test code behavior and proposes a tool called MeteoR. This tool comprises an IDE plugin to detect issues that may arise during test code refactoring, reducing the effort required to perform evaluations. A preliminary assessment was conducted to validate the tool and ensure the proposed test code refactoring approach is adequate. By analyzing not only the mutation score but also the generated mutants in the pre- and post-refactoring process, results show that the approach is capable of checking whether the behavior of the mutants remains unchanged throughout the refactoring process. This proposal represents one more step toward the practice of test code refactoring. It can improve overall software quality, allowing developers and testers to safely refactor the test code in a scalable and automated way.

Keywords: software engineering; test code refactoring; test smells; mutation testing

1. Introduction

Refactoring software code is an essential area of software engineering that requires safety nets of protection to avoid degradation of application behavior after code correction. As declared by Parsai et al. [1], refactoring is not an activity that only concerns the application code but also actively involves the test code. Meszaros [2] states that tests, when refactored, should not change external behavior, only changes in internal design.

When refactoring the application code, automated test code serves as a safety net to ensure the quality of the production code. However, if the test suite code itself undergoes refactoring, the safety net it provides is lost [3]. Some researchers have suggested using mutation testing to protect the refactored test from changing its behavior[1]. Mutation testing is injections of intentional failures performed in the application code to validate the behavior of the tests [4]. A study conducted by Parsai et al. [1] verified that mutation testing allows identifying: 1) changes in the behavior of the refactored test code; and 2) which part of the test code was improperly refactored. Specifically, when evaluating changes in behavior, Parsai et al. [1] relies on comparing mutation scores before and after refactoring. Moreover, although Parsai et al. [1] have created a mutation testing tool named LittleDarwin[1], it focuses only on performing the mutation testing and does not provide any feature to support test code refactoring. So, even if mutation testing was pointed out as an alternative to providing safety for refactoring test code, no tool implemented an automated analysis based on that to evaluate the refactored test behavior.

[1] https://littledarwin.parsai.net/

 doi:10.20944/preprints202310.0969.v1

In this paper, we propose the development of a tool called MeteoR (**M**utant-bas**e**d **te**st c**o**de **R**efactorings) that simplifies the evaluation of the behavior of the test mutation during the test code refactoring. Designed as an Eclipse[2] plugin, this tool can significantly enhance safety on test code refactoring by combining the IDE environment, mutant testing, and analysis/reporting tools. In addition to the previous work, this work proposes the analysis of each mutation individually to provide a more thorough assessment of the refactored test behavior.

This study presents three main contributions: 1) an extension of the approach introduced by Parsai et al. [1] that incorporates an in-depth automated analysis of test mutation behavior; 2) a proposal of a tool conception to speed up the refactoring analysis of the test code called MeteoR; and 3) a preliminary feasibility assessment to validate the functional aspects of the proposed tool based on the extended approach.

The remainder of this paper is organized as follows: Section 2 provides a leveling of knowledge by presenting the theoretical aspects necessary to understand the research question and the proposed approach. Section 3 presents the related works in the literature. Next, in Section 4, a state-of-the-art test refactoring tool concept is described in detail. Section 5 presents a preliminary assessment of the proposed tool and approach for evaluating test code refactoring. Section 6 discusses the results obtained from this preliminary evaluation. Finally, in Section 7, the authors conclude on the study results and provide perspectives for future work.

2. Background

2.1. Test Code Refactoring

As Meszaros [2] states, test code refactoring differs from application code refactoring because there are no "automated tests" for automated tests. If a test fails after the test code refactoring, it is challenging to verify whether a failure occurred due to an error introduced during the refactoring. Similarly, if the test passes after the test code refactoring, it is difficult to guarantee that it will fail when it is expected to fail.

This goes hand in hand with test automation because it is very complicated to refactor the test code without having a safety net that guarantees that automated tests do not break during their redesign states [2].

Guerra and Fernandes [5] and Meszaros [2] state that test code refactoring differs from application code refactoring. According to Guerra and Fernandes [5], when the change is applied in the test code, the concept of the behavior of the test code is different from the behavior of the application code, so it makes no sense to create tests to verify the behavior of the test code. That is, the way to evaluate the application's behavior differs from how to assess the test behavior.

Test code refactoring can be motivated by: 1) the elimination of bad smells in the test code, or test smells[3]; and 2) the need to refactor the application code, which may involve adapting the test code.

This work sheds light on condition (1) since the refactoring of the application code (2) and the subsequent refactoring of the test codes are a situation that results in different sets of mutants, making the comparison much more difficult [1]. To address the second situation, Parsai et al. [1] suggests dividing the refactoring into two parts. First, the application code refactoring, with the execution of the tests, ensures that the application's behavior has not changed. Second, in test code refactoring, it is possible to apply the suggested and detailed concept of this study as described in Section 5.

---

[2] https://eclipseide.org

[3] The term "test smells" was coined by Van Deursen et al. [6] as a name for symptoms in the test code that possibly indicate a deeper problem.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

2.2. Mutation Testing

Mutation testing is the process of injecting faults into the application source code. This field of research dates back to the early 1970s when Lipton proposed the initial concepts of mutation in a paper entitled "Fault Diagnosis of Computer Program" [4]. It is performed as follows: first, a faulty application version is created by injecting a purposeful fault (mutation). One operator (mutation operator or mutator) transforms a specific part of the code. After the faulty versions of the software (mutants) have been generated, the test suite is run on each of these mutants. According to Offutt and Untch [4], mutation testing objectively measures the suitability (effectiveness) level of analyzed test sequences, called score mutation. It is calculated as the ratio of dead mutants (the ones detected by the test suite) over the total number of non-equivalent ones. Equivalent mutants are semantically equivalent to the original program. Thus, they can not be killed by any test case. A manual process usually shows equivalence during the execution of test cases.

This score quantifies the effectiveness of the test. Mutants not detected by the tests provide valuable information for improving the test set by identifying areas that require additional tests. In this way, mutation testing guides the improvement of a test suite. It is up to the developer to analyze the test logs and reports to validate whether the survival mutants are subject to correction. Finally, the developer refactors the test code to ensure, in a new round of mutation test execution, that previously surviving mutants have been killed.

3. Related Work

The literature review technique known as "Snowballing" [7] was applied to retrieve the most critical articles on the subject. Some searches merged the "test refactoring" and the "test mutation" strings in this work.

There is a vast body of literature on the subject of test mutants or test refactoring. However, the objective of our research is not to use mutation tests directly to evaluate the quality of test suites. Instead, the goal is to employ mutation tests according to the Parsai et al. [1] approach to measuring the behavior and effectiveness of the refactored test code.

Pizzini [8] primarily relies on instrumentation. This involves instrumenting both the system under test (SUT) and its tests to detect the entry and exit points of methods, modifications in SUT class attributes, and selection and repetition structures. The resulting instrumentation enables the creation of a code execution tree, which can be used to identify the behavior of the SUT and its tests. During this step, the syntactic and semantic analysis of the SUT and test code is used to identify specific points of the code, such as object creation and modifications to the internal states of created objects. It is worth noting that this approach may require significant effort to instrument all the code, which could discourage some developers. Nevertheless, it provides full observability of test and application behavior after refactoring.

Bladel and Demeyer [3]'s approach involves constructing a test behavior tree using a technique inspired by symbolic execution. This tree is constructed for both the pre- and post-refactoring test cases, and a comparison between them is made to determine whether the test behavior has been preserved. The similarity between the two trees is crucial to preserving behavior.

Regarding tools that can support behavior preservation, AlOmar et al. [9] argue that there is significant potential to propose and improve automated tools, not only in the context of test code refactoring but also in software refactoring in a more general sense.

Based on the related works, three primary categories of tools were identified:

Test Code Refactoring Tool: To verify changes in test behavior, Parsai et al. [1] highlights the importance of using mutation testing to check for changes in test behavior. In contrast, Bladel and Demeyer [3] proposes a distinct approach using symbolic execution. A tool called T-CORE (Test Code Refactoring Tool) generates a report indicating whether test behavior has changed after execution. An alternative tool proposal, SafeRefactor, introduced by Soares [10], provides valuable perspectives on assisting developers during refactoring, despite not being a test code refactoring tool. Aljawabrah

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

et al. [11] proposes a tool to facilitate the visualization of test code traceability (TCT - Test-to-Code Traceability), which can assist in the process of refactoring test code.

Test Bad Smell Detection Tool: According to van Bladel and Demeyer [12], there is a limited number of bad smell detection tools for testing. Peruma et al. [13] propose a tool called TSDetect[4] (Test Smell Detector) to detect and address bad smells in code. This tool reads a .csv configuration file containing a list of classes to be checked and identifies any bad smells. Figure 1 presents the high-level architecture of TSDetect.

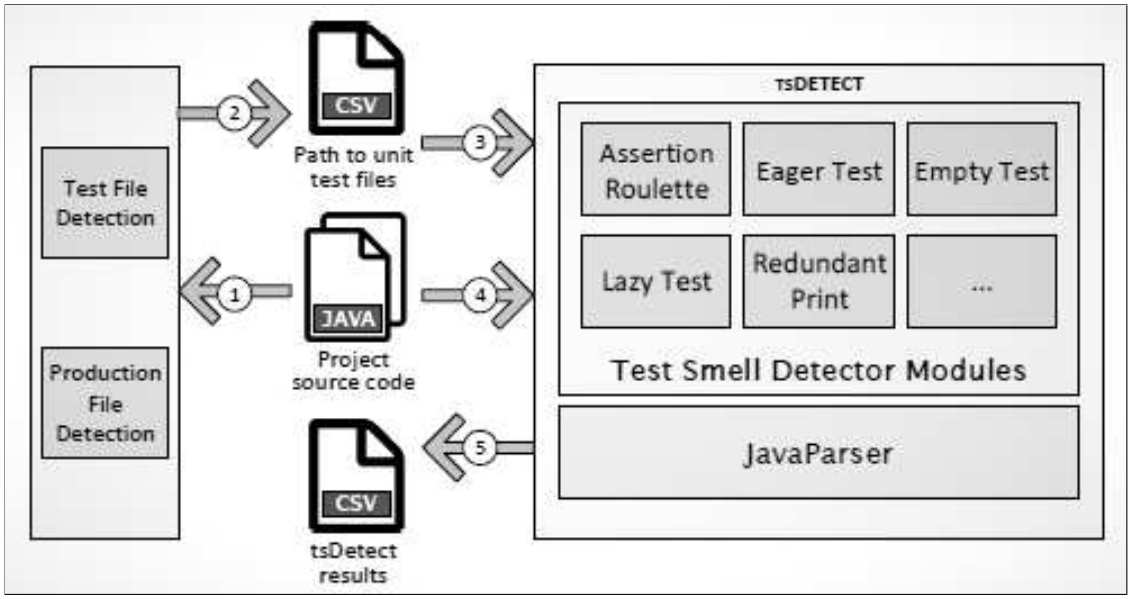

Figure 1. High-level architecture of TSDetect tool [13].

Marinke et al. [14] proposed an architecture called EARTC (Extensible Architecture for Refactoring Test Code). A plugin called Neutrino [14] was developed for the Eclipse IDE to assist in refactoring the test code and the identification of bad smells, as seen in Figure 2.

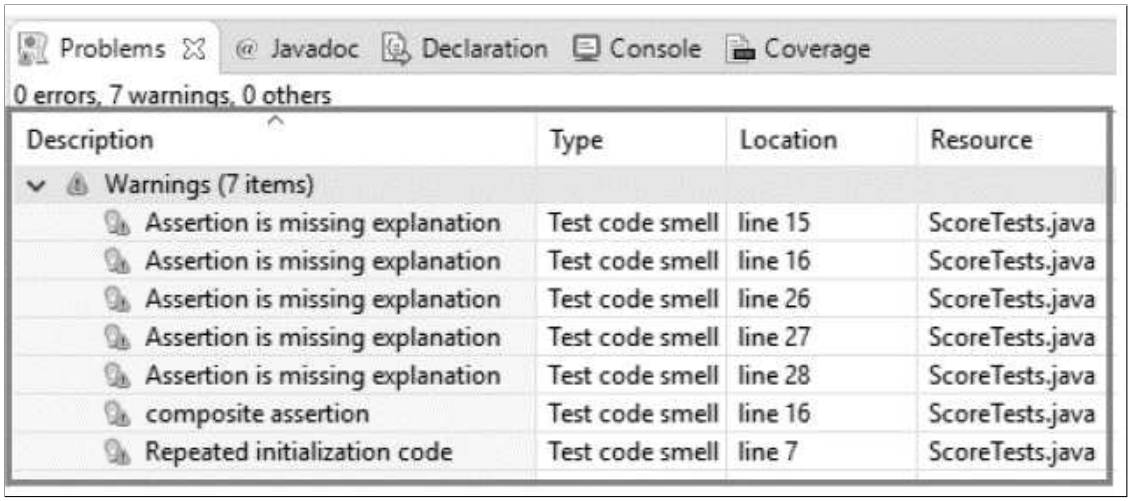

Figure 2. Eclipse Neutrino plugin and the EARTC architecture idenfitying test code smells [14].

[4] https://testsmells.org/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

Mutation Testing Tools: Papadakis et al. [15], reported an increase in mutation testing tools developed between 2008 and 2017, with 76 tools created for various software artifacts. While most of these tools target implementation-level languages, the C and Java programming languages are the primary focus of mutation testing tools at the implementation level. According to Singh and Suri [16], Java has the highest number of mutation testing tools among different languages. These tools include MuJava, PIT (or PITest), Judy, Jester, Jumble, and Bacterio. As noted by Monteiro et al. [17], PIT is a widely-used tool for research purposes and has also gained traction in the industry.

4. MeteoR: An Integrated Tool Proposal for Test Code Refactoring

We have compiled a list of nine main stages (functionalities) that can form an integrated tool model for test code refactoring, covering all the requirements gathered to this task [3,9,13,14,17–20].

The primary stages of MeteoR are presented in Figure 3 as a sequential arrangement of test code refactoring activities, while a detailed explanation of the roles played by each stage can be found in Table 1.

Since our tool proposal is currently in the development phase, we are focusing specifically on implementing and integrating stages 4, 5, and 6, which are centered on validating changes in test behavior. Our research group is also working on the other stages to develop the whole workflow of MeteoR.

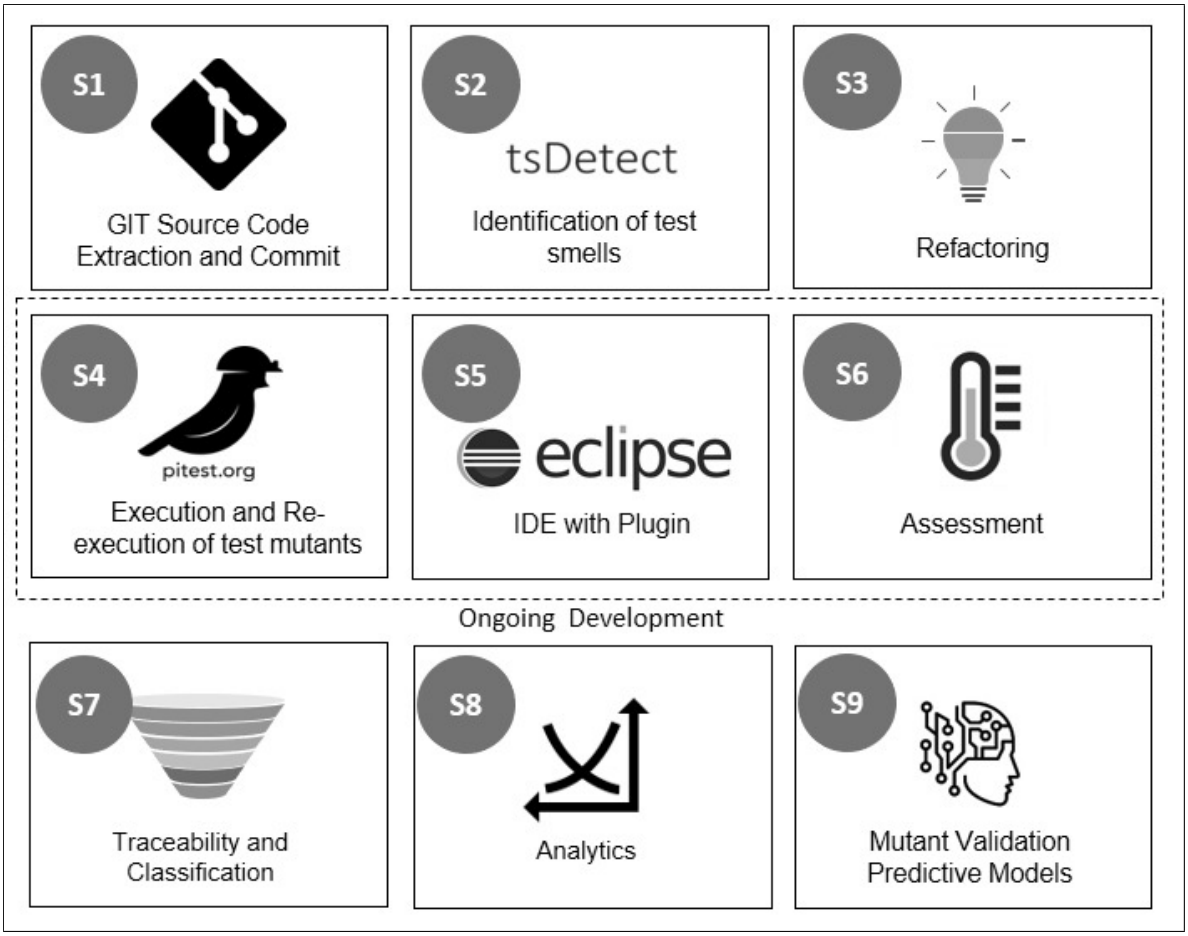

Figure 3. A holistic view (workflow) of the MeteoR's main stages (S_n). The dashed box indicates the ongoing stages addressed in this paper.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

Table 1. Overview of MeteoR tool's main stages.

#	Stage	Role
S1	Source code extraction and commit	Connectivity with the *GIT* tool.
S2	Identification of test smells	Integration with any pre-existing tool in order to provide quick verification of test codes that need improvement.
S3	Refactoring	Perform the refactoring in an assisted way, trying to solve the test code quality gaps with automated fix suggestions.
$S4_a$	Execution of mutation testing	Use of a tool to generate test mutants and run mutation testing to assess the quality of project tests. Its first run will serve as a test baseline.
$S4_b$	Reexecution of the mutation testing	Re-run the mutation testing scenarios under the refactored test code and compare the new results with the baseline result.
S5	IDE Plugin Integration	Orchestrate test mutation runs, collecting data, performing analysis, and generating results reports.
S6	Assessment	Have mutants been modified? Comparing results will provide those answers. If there was no change in the state of the mutants, then the refactoring was done successfully.
S7	Traceability and Classification	Catalog and identification of test mutants (killed and surviving). Improve traceability in the refactoring cycles.
S8	Analytics	Evaluate data and generate views to monitor the evolution of test mutants throughout code refactorings.
S9	Mutant Validation Predictive Models	Application of *Machine Learning* and *AI* tools to obtain insights, refinement, prediction, and selection of test mutants.

Legend: S# = Stage in Figure 3

4.1. The MeteoR Workflow

MeteoR workflow starts with cloning a project from a GIT [5] repository to apply the test code refactoring (Stage 1 – S1 – Figure 3).

Next, a list of identified bad smells in test code is prepared using a tool, such as TSDetect (S2), or through the developer's own experience in recognizing the types of bad smells that need to be corrected. In future versions, it is planned that S1 and S2 can act in silent mode, producing a backlog that serves as the basis for the next stage.

The bad smells identified in the test code are then analyzed using a third tool to indicate the best possible fixes (S3). Here, the PITest tool (S4) is called to generate the initial report of the test mutants, which will serve as a baseline for comparison after refactoring.

The refactoring process is carried out using Eclipse IDE. (S5) due to its wide variety of plugins, including the PITest tool and JUnit[6]. Once refactored, unit tests are run in either JUnit[6] or TestNG[7] to ensure 100% execution success.

---

[5] AlOmar et al. [9] underline the importance of integrating this kind of tool with control version systems such as Git or Subversion.

[6] https://junit.org/junit5/

[7] https://testng.org/doc/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

The PITest tool (S4) is called again to generate the final view of the test mutants. Comparing the results of the two mutant test runs will determine whether there has been a change in the behavior of the test code (S6).

Throughout the workflow, all the generated data is traceable and classified (S7) and provide rich indicators that can be reported in an analytics dashboard (S8) or used for prediction models (S9).

4.2. *MeteoR Software Components*

To implement the stages S4, S5, and S6 outlined earlier, four essential software components have been derived and are currently being developed. A high-level depiction of the interactions between these components can be seen in Figure 4, while Table 2 provides a detailed overview of the objectives of each component.

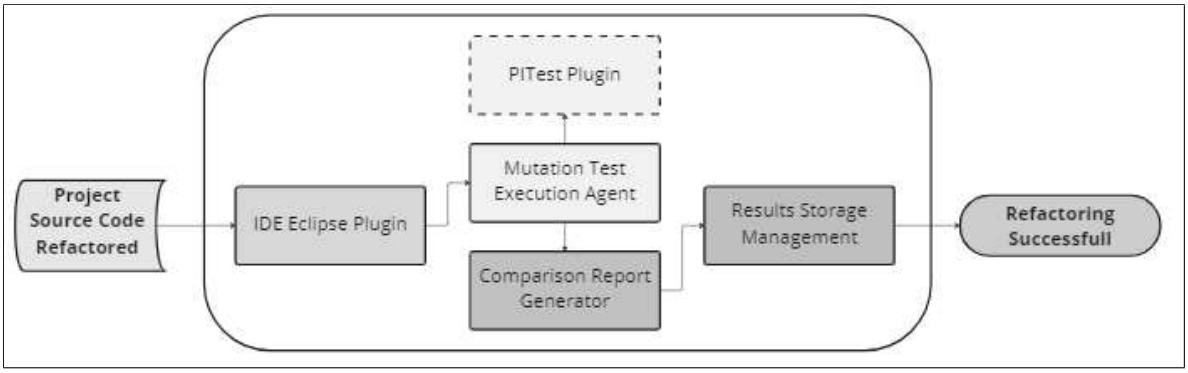

Figure 4. The four key software components of the MeteoR.

Table 2. Description of MeteoR's main software components.

Component	Description
IDE *Eclipse* Plugin	An IDE plugin implementation provides a familiar environment for developers to perform refactoring and analysis tasks within a single tool. This approach will allow for a streamlined development experience, improving productivity and reducing the possibility of errors.
Mutation Testing Execution Agent	Component that calls the PITest tool to generate and run mutant testing.
Comparison Report Generator	The data should be compared pre- and post-refactoring from the test mutants and a report generated indicates whether there was any change in the test behaviors. This report not only include a one-to-one comparison of mutations, but it will also evaluate the mutation scores.
Results Storage Management	Local storage to maintain the results and other artifacts.

In Figure 4, the sequential order of the software components invoked is shown, starting from the source code of the refactored project and ending with a successful refactoring. In the case a refactoring process was unsuccessful, the developer should review the refactored test code and either roll back the changes or redo the refactoring with the necessary corrections. Figure 5 shows the component diagram that illustrates the integration between each component.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

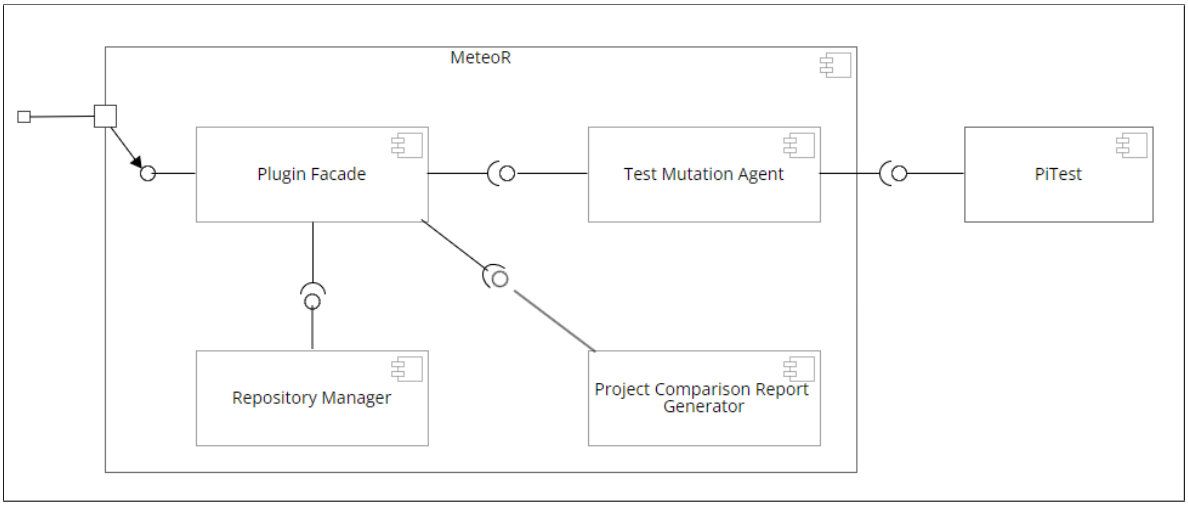

Figure 5. MeteoR's UML component diagram.

5. Preliminary evaluation of the proposed tool

Before implementing MeteoR, we manually reproduced all the steps of the proposed approach to verify if the tool can achieve the expected verification of test code behavior. That is one of the reasons for this preliminary evaluation.

To verify the correctness of a test code refactoring, we employ the approach proposed by Parsai et al. [1] to determine whether it induces changes in the behavior of test mutants. Conversely, an incorrect test code refactoring should result in test mutants' behavior changes.

An essential difference between our study and the study conducted by Parsai et al. [1] is that we analyzed individual mutations before and after refactoring rather than relying solely on comparing mutation scores to assess test code behavior. Subsection 5.4 contains a sample table that compares all mutants generated before and after the test code refactoring activity.

5.1. Methodology

Our assessment methodology involves two distinct procedures to evaluate the proposed approach. Here, the concept of positive and negative tests is utilized.

The positive test procedure is executed to validate the ability of a system or application to perform correctly under valid input conditions. In our context, whether proper test code refactoring was performed.

The negative test procedure involves testing the application by inputting invalid or inappropriate data sets and evaluating if the software responds as intended when receiving negative or unwanted user inputs. In the present context, the focus was on verifying whether the approach could effectively handle an inappropriate refactoring of the test code and respond accurately. For the case study, we selected the *Apache Commons-csv*[8] project and applied refactorings to the CSVRecordTest class, which was then subjected to both evaluation procedures.

In the positive test procedure, one or more test classes with bad smells are selected, and mutation testing is performed in the related application classes using the PITest default operators[9]. The results are then recorded to establish a baseline, and the test code methods are properly refactored. Mutation testing is repeated, and results are recorded and tabulated. The behavior of individual mutants is then

[8] https://github.com/apache/commons-csv
[9] https://pitest.org/quickstart/mutators/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

validated line by line to determine whether the refactoring was successful, which means that there were no changes in the behavior of the test mutants.

During the negative test procedure, the case study is restored to its initial state, and improper refactoring is performed. This refactoring affects the test's behavior, but does not affect the test execution result, meaning the test must still pass. Subsequently, mutation testing is conducted again, and a comparison with the baseline must show changes in the behavior of the test mutants, indicating improper refactoring.

5.2. Positive Test Procedure Execution

5.2.1. Assessing the bad smells before test code refactoring

Table 3 shows the report with 13 bad-smell tests of the Assertion Roulette type, as addressed by Soares et al. [21]. Assertion Roulette is a test smell that makes it difficult to identify which assertion caused a test run failure.

Table 3. TSDetect report – Assertion Roulette bad smells detected in test files.

Relative test file path	Number of methods	Assertion roulette
CSVRecordTest.java	31	13
CSVDuplicateHeaderTest.java	4	2
IOUtilsTest.java	1	0

5.2.2. First run of pre-refactoring mutation testing

Table 4 presents the report of the first run of the mutation testing (pre-refactoring).

Table 4. PITest coverage report – Mutants generated during the first run of PITest tool.

Class	Line Coverage	Mutation Coverage (mutation score)	Test Strength (mutation score)
CSVRecord	49/49	47/48 (0.9791)	47/48 (0.9791)

5.2.3. Test code refactoring

A test code refactoring was applied in the CSVRecordTest class. Specifically, the test code was refactored to extract the grouped assertions, mitigating the risk of Assertion Roulette. Figure 6 (a) presents this type of test smell and Fig 6 (b) – (lines 4, 9, 14, and 19) – its fixed test code refactoring.

a. Test code with bad smells.

```
1  @Test
2  public void testToString() {
3    assertNotNull (recordWithHeader.toString());
4    assertTrue (recordWithHeader.toString().contains("comment="));
5    assertTrue (recordWithHeader.toString().contains("recordNumber="));}
6    assertTrue (recordWithHeader.toString()
7  .contains("values="));}
8  }
```

b. Test code fixed.

```
1  @Test
2  public void testHeaderEmptyToString(){
3      assertNotNull(recordWithHeader.toString(),
4          " Validate if record null returns empty. ");
5  }
6  @Test
7  public void testHeaderCommentToString(){
8      assertTrue(recordWithHeader.toString().contains("comment="),;
9          " Validate if records contains comment. ");
10 }
11 @Test
12 public void testHeaderRecordNumber(){
13     assertTrue(recordWithHeader.toString().contains("recordNumber="),
14         "Validate if records contains recordNumber." );
15 }
16 @Test
17 public void testRecordValues(){
18     assertTrue(recordWithHeader.toString().contains("values="),
19         "Validate if records contains values." );
20 }
```

Figure 6. CSVRecordTest class before and after proper refactoring.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

5.2.4. Mutation testing run

Table 5 displays the report of the second execution of mutation testing (post-refactoring).

Table 5. PITest coverage report – Mutants generated during the second run of PITest tool.

Class	Line Coverage	Mutation Coverage (mutation score)	Test Strength (mutation score)
CSVRecord	49/49	47/48 (0.9791)	47/48 (0.9791)

Observing Table 4, it can be seen that performing a proper refactoring of the test code and removing bad smells had no effect on the mutation score.

5.3. Negative Test Procedure Execution

5.3.1. Test Code Refactoring

During the negative test, bad smells in the test code were identified without relying on a bad smell detection tool like TSDetect. Specifically, we found that the selected tests exhibit non-standardized code for checking the items on the list, resulting in duplicated and non-uniform code. As shown in Figure 7 (a), the same type of check is performed in three different ways. This can make the code difficult to modify and maintain, leading to errors and decreased productivity. Therefore, it is crucial to refactor the test code to eliminate these bad smells and improve the overall quality of the codebase.

a. Bad smells in test code

```
1  @Test
2  public void testToListFor() {
3      int i = 0;
4      for (final String value : record.toList()) {
5          assertEquals(values[i], value) ;
6          i++ ;
7      }
8  }
9  @Test
10 public void testStream() {
11     final AtomicInteger i = new AtomicInteger();
12     record.stream().forEach(value -> {
13         assertEquals(values[i.get()], value) ;
14         i.incrementAndGet() ;
15     });
16 }
17 @Test
18 public void testToListForEach() {
19     final AtomicInteger i = new AtomicInteger();
20     record.toList().forEach(e -> {
21         assertEquals(values[i.getAndIncrement()] , e);
22     });
23 }
```

b. Test code fixed.

```
1  @Test
2  public void testToListFor() {
3      for(int i=0; i < record.size(); i++){
4          assertValuesAndIncrement (0, record.toList().get(0));
5      }
6  }
7  @Test
8  public void testStream() {
9      record.stream().forEach(value -> {
10         assertValuesAndIncrement (0, record.toList().get(0));
11     });
12 }
13 @Test
14 public void testToListForEach() {
15     record.toList().forEach(e -> {
16         assertValuesAndIncrement (0, record.toList().get(0));
17     });
18 }
19
20 public void assertValuesAndIncrement(int i, String value){
21     assertEquals(values[i], value);
22     ++i;
23 }
```

Figure 7. CSVRecordTest class before and after improper refactoring.

It shall be emphasized that the intentional correction of this verification was carried out incorrectly, as evidenced in Figure 7 (b), lines 4, 10, 16, and 22.

Here, a situation is simulated in which the developer improperly sets the index when it comes to iterating through the elements of the lists.

In this situation, the developer supposedly forgot to pass the value of the variable i that controls the iterations as a parameter to the assertValuesAndIncrement method.

Setting the value to 0 (zero), he/she changed the behavior of the test method, as it stops comparing all the elements and performs the same comparison several times, always with the first element in the list.

Although the test was incorrectly refactored, it passed successfully during the initial run. However, the developer significantly modified the validation behavior, which is expected to alter the behavior

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

of the mutants during subsequent mutation testing. This confirms that the developer's procedure contains an error and indicates that the refactored test is less effective than the previous non-refactored test. This way, the new code offers poorer validation than the previous version.

5.3.2. Mutation Testing Run

Table 6 shows the report of the second session of the mutation testing (post-refactoring). As expected, the incorrect test code refactoring caused a change in the mutation score, demonstrating that the refactoring work was unsuccessful.

Table 6. PITest coverage report - mutants generated during the second run of PITest tool after improper refactoring.

Class	Line Coverage	Mutation Coverage (mutation score)	Test Strength (mutation score)
CSVRecord	49/49	46/48 (0.9583)	46/48 (0.9583)

5.4. Mutation Data Compilation and Comparison

To gain a more detailed understanding of mutation behavior, we compiled all mutant data from pre- and post-refactoring tests in worksheets[10]. In doing so, we extended the Parsai et al. [1] approach, which focused only on mutation score.

Table 7 provides consolidated data on test mutations for the CSVRecord class, grouped by mutation operators and tool executions. Upon analyzing the data, we can observe the expected changes in mutation behavior between the first and second runs, following an improper test code refactoring, highlighted in Table 7, row (8). To further explore the results, we can examine the mutators (mutation operators) and the corresponding line codes that result in the observed behavior changes, as shown in Table 8.

Table 7. Mutation testing data from CSVRecord (grouped by mutation operator).

Mutator	1st Run		2nd Run	
	Killed	Survived	Killed	Survived
Changed conditional boundary	4	0	4	0
Negated Conditional	15	1	15	1
Removed call to	1	0	1	0
Replaced boolean return with false	5	0	5	0
Replaced boolean return with true	5	0	5	0
Replaced int return with 0	1	0	1	0
Replaced long return with 0	2	0	2	0
Replaced return value with ""	5	0	4	1
Replaced return value with Collections.emptyList	1	0	1	0
Replaced return value with null	8	0	8	0
Grand Total	47	1	46	2
Mutation Coverage	47/48		46/48	
Mutation Score	0.9791		0.9583	

Analyzing the detailed behavior of each mutant pre- and post- refactoring strengthens the original approach with this additional step. In other words, it ensures that the refactored test code not only achieved similar mutation scores but also preserved the same mutation structure. This situation is

[10] available at: https://gitlab.com/meteortool/assessment/-/blob/main/data/Comparativo.xlsx

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 October 2023 doi:10.20944/preprints202310.0969.v1

critical in cases of improper refactoring, where changes in the mutation structure can indicate potential issues in the test behavior. For this reason, this is a major contribution to this article since the Parsai et al. [1] analysis does not go into this level of detail.

Table 8. Detailed analysis of pre- and post-test code refactoring.

Line	First Run	Second Run	Unchanged?
287	1. replaced int return with 0 for **org/apache/commons/csv/CSVRecord**::size → KILLED	1. replaced int return with 0 for **org/apache/commons/csv/CSVRecord**::siz → KILLED	TRUE
297	1. replaced return value with null for **org/apache/commons/csv/CSVRecord** → KILLED	1. replaced return value with null for **org/apache/commons/csv/CSVRecorc** → KILLED	TRUE
310	1. replaced return value with Collections.emptyList for **org/apache/commons/csv/CSVRecord**::**toList** → KILLED	1. replaced return value with Collections.emptyList for **org/apache/commons/csv/CSVRecord**::**toList** → KILLED	TRUE
322	1. replaced return value with null for **org/apache/commons/csv/CSVRecord** → KILLED	1. replaced return value with null for **org/apache/commons/csv/CSVRecorc** → KILLED	TRUE
333	1. replaced return value with "" for **org/apache/commons/csv/CSVRecord**:: → KILLED	1. replaced return value with "" for **org/apache/commons/csv/CSVRecord**:: → SURVIVED	FALSE
344	1. replaced return value with null for **org/apache/commons/csv/CSVRecord** → KILLED	1. replaced return value with null for **org/apache/commons/csv/CSVRecorc** → KILLED	TRUE

6. Result Analysis and Discussions

Results obtained in this study reinforce what Parsai et al. [1] have highlighted: mutant testing is a safety net to guarantee a correct test code refactoring.

To be able to measure whether the test code refactoring was successful, it was necessary to produce tables for viewing and comparing the data of the behavior of the mutants from the pre- and post-refactoring tests in the two refactoring sessions (proper and improper sections).

In scenarios where test code refactoring involves multiple classes and test methods, controlling and monitoring the mutation data can be challenging, which can differ enormously in each section of test code refactoring. That is major evidence of the improvement in the work of analysis and refactoring check our approach is capable of.

Upon comparing Table 5, which displays the mutation score after proper refactoring of the test code, with Table 6, which displays the mutation score after improper refactoring of the test code, it becomes evident that all mutations remain unchanged given the correct refactoring when comparing the results of the pre- and post-refactoring.

However, in incorrect refactoring, as shown in Subsection 5.3, one can notice in Table 8 (row 5) the difference in the behavior of the mutants, which means an error in the refactoring process.

In addition, this paper has improved the analysis activity, comparing not only the mutation score but also all the mutants classified in the pre- and post-refactoring tables. This approach allows us to validate each mutation and detect mutant behavior more accurately, as stated in the previous paragraph.

Although we cannot definitively state that this method eliminates the threat of the masking effect, as noted by Parsai et al. [1], this allows us to ensure that the refactored code maintains the same mutation structure in addition to achieving similar mutation scores. By closely examining each mutation behavior, it is possible to detect and resolve any potential issues that may arise during the refactoring process, thereby improving the overall reliability and robustness of the code.

Developing an integrated tool to facilitate the implementation of the concepts presented in this study is viable and necessary to assist developers in performing test code refactoring. Noteworthy data is presented below, emphasizing the benefits of automating this type of work. By automating the analysis process, efficiency can be improved, and the potential for errors in test refactoring can be reduced, ultimately resulting in higher-quality code.

In the present study, each execution of the mutation tests required approximately 12 minutes to validate eight application classes. That is, it took around 36 minutes to run the mutation testing on its two executions. In total, 782 mutations were generated, and 5,231 tests were performed, with an average of 6.69 tests per mutation.

 doi:10.20944/preprints202310.0969.v1

Consequently, several key lessons learned and opportunities for consideration during the implementation of the integrated tool have been identified, including:

1. Ensure to isolate the tests only for the relevant classes within the scope of the refactoring.
2. Evaluate the possibility of improving the parallelism to accelerate the generation and treatment of mutants.
3. Consider preventing the modification of productive classes while the test classes are refactored.
4. Reuse as much as possible the previously generated mutants; it is believed that there will be a decrease in the computational cost of changing the code for generating mutations and compiling the project, that is, evaluating how these mutants can be maintained so that a complete build is not necessary for each new execution of the mutation testing.

7. Conclusion

This study highlights the importance of developing solutions that simplify the observation of mutation test behavior in test code to confirm the quality of refactorings.

The investigation achieved its three primary objectives, as evidenced by the results. First, we extended the Parsai et al. [1]'s approach by incorporating an in-depth automated analysis of test mutation behavior. Second, a tool conception was presented to speed up the test code refactoring based not only on mutation score but also analyzing the detailed behavior of each mutant pre- and post-refactoring. To address these issues, an integrated tool concept, called MeteoR, was proposed to refactor the test code and to analyze its quality cohesively. Finally, we evaluated the feasibility of the expanded approach by conducting a preliminary assessment that simulates some of the tool's capabilities. The assessment validated the approach and revealed that MeteoR is able to verify problems in the test code refactoring process.

This paper focused on addressing the critical challenges associated with mutant testing analysis and refactoring, accelerating the refactoring of the test code, and ensuring its robustness. In summary, this study has made progress toward proposing a tool for specifically monitoring the behavior of test code refactoring.

In the future, providing tools that can perform refactoring by integrating test code correction and behavior verification autonomously would be crucial to avoid the need for additional human effort and rework to analyze the correctness of the refactoring activity. Moving forward, the next steps of this research involve finalizing the stages S4, S5, and S6, testing, and publishing a stable version of the tool for community use. The plan is to establish a Continuous Integration (CI) pipeline with the necessary DevOps mechanisms and best practices, ensuring the efficient delivery of the tool.

Author Contributions: All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) grant number 2023/04581-0.

Data Availability Statement: The data presented in this study are openly available in the gitlab repository, in the following link: https://gitlab.com/meteortool/assessment/-/blob/main/data/Comparativo.xlsx.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

IDE	Integrated Development Environment
SUT	System Under Testing
TCT	Test-to-Code Traceability
UML	Unifed Modeling Language

References

1. Parsai, A.; Murgia, A.; Soetens, Q.D.; Demeyer, S. Mutation Testing as a Safety Net for Test Code Refactoring. In Proceedings of the Scientific Workshop Proceedings of the XP2015, New York, NY, USA, 2015; XP '15 workshops. https://doi.org/10.1145/2764979.2764987.
2. Meszaros, G. *xUnit Test Patterns: Refactoring Test Code*; Pearson Education, 2007.
3. Bladel, B.v.; Demeyer, S. Test Behaviour Detection as a Test Refactoring Safety. In Proceedings of the Proceedings of the 2nd International Workshop on Refactoring, New York, NY, USA, 2018; IWoR 2018, p. 22–25. https://doi.org/10.1145/3242163.3242168.
4. Offutt, A.J.; Untch, R.H. *Mutation 2000: Uniting the Orthogonal*; Springer US: Boston, MA, 2001; pp. 34–44. https://doi.org/10.1007/978-1-4757-5939-6_7.
5. Guerra, E.M.; Fernandes, C.T. Refactoring Test Code Safely. In Proceedings of the International Conference on Software Engineering Advances (ICSEA 2007), 2007, pp. 44–44. https://doi.org/10.1109/ICSEA.2007.57.
6. Van Deursen, A.; Moonen, L.; Van Den Bergh, A.; Kok, G. Refactoring test code. In Proceedings of the Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP); Marchesi, M., Ed., 2001.
7. Wohlin, C. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In Proceedings of the Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, New York, NY, USA, 2014; EASE '14. https://doi.org/10.1145/2601248.2601268.
8. Pizzini, A. Behavior-based test smells refactoring : Toward an automatic approach to refactoring Eager Test and Lazy Test smells. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2022, pp. 261–263. https://doi.org/10.1145/3510454.3517059.
9. AlOmar, E.A.; Mkaouer, M.W.; Newman, C.; Ouni, A. On Preserving the Behavior in Software Refactoring: A Systematic Mapping Study. *Inf. Softw. Technol.* **2021**, *140*. https://doi.org/10.1016/j.infsof.2021.106675.
10. Soares, G. Making program refactoring safer. In Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2010, Vol. 2, pp. 521–522. https://doi.org/10.1145/1810295.1810461.
11. Aljawabrah, N.; Gergely, T.; Misra, S.; Fernandez-Sanz, L. Automated Recovery and Visualization of Test-to-Code Traceability (TCT) Links: An Evaluation. *IEEE Access* **2021**, *9*, 40111–40123. https://doi.org/10.1109/ACCESS.2021.3063158.
12. van Bladel, B.; Demeyer, S. Test Refactoring: a Research Agenda. In Proceedings of the Proceedings SATToSE, 2017.
13. Peruma, A.; Almalki, K.; Newman, C.D.; Mkaouer, M.W.; Ouni, A.; Palomba, F. TsDetect: An Open Source Test Smells Detection Tool. In Proceedings of the Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 2020; ESEC/FSE 2020, p. 1650–1654. https://doi.org/10.1145/3368089.3417921.
14. Marinke, R.; Guerra, E.M.; Fagundes Silveira, F.; Azevedo, R.M.; Nascimento, W.; de Almeida, R.S.; Rodrigues Demboscki, B.; da Silva, T.S. Towards an Extensible Architecture for Refactoring Test Code. In Proceedings of the Computational Science and Its Applications – ICCSA 2019; Misra, S.; Gervasi, O.; Murgante, B.; Stankova, E.; Korkhov, V.; Torre, C.; Rocha, A.M.A.; Taniar, D.; Apduhan, B.O.; Tarantino, E., Eds., Cham, 2019; pp. 456–471.
15. Papadakis, M.; Kintis, M.; Zhang, J.; Jia, Y.; Traon, Y.L.; Harman, M. Chapter Six - Mutation Testing Advances: An Analysis and Survey; Elsevier, 2019; Vol. 112, *Advances in Computers*, pp. 275–378. https://doi.org/https://doi.org/10.1016/bs.adcom.2018.03.015.
16. Singh, D.; Suri, B. Mutation testing tools- An empirical study. In Proceedings of the Third International Conference on Computational Intelligence and Information Technology (CIIT 2013), 2013, pp. 230–239. https://doi.org/10.1049/cp.2013.2596.
17. Monteiro, R.; Durelli, V.H.S.; Eler, M.; Endo, A. An Empirical Analysis of Two Mutation Testing Tools for Java. In Proceedings of the Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing, New York, NY, USA, 2022; SAST '22, p. 49–58. https://doi.org/10.1145/3559744.3559751.
18. Offutt, J. A mutation carol: Past, present and future. *Information and Software Technology* **2011**, *53*, 1098–1107. Special Section on Mutation Testing, https://doi.org/https://doi.org/10.1016/j.infsof.2011.03.007.

19. Zhu, Q.; Zaidman, A.; Panichella, A. How to kill them all: An exploratory study on the impact of code observability on mutation testing. *Journal of Systems and Software* **2021**, *173*, 110864. https://doi.org/https://doi.org/10.1016/j.jss.2020.110864.
20. Ojdanic, M.; Soremekun, E.; Degiovanni, R.; Papadakis, M.; Le Traon, Y. Mutation Testing in Evolving Systems: Studying the Relevance of Mutants to Code Evolution. *ACM Trans. Softw. Eng. Methodol.* **2022**. https://doi.org/10.1145/3530786.
21. Soares, E.; Ribeiro, M.; Amaral, G.; Gheyi, R.; Fernandes, L.; Garcia, A.; Fonseca, B.; Santos, A. Refactoring Test Smells: A Perspective from Open-Source Developers. In Proceedings of the Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing, New York, NY, USA, 2020; SAST 20, p. 50–59. https://doi.org/10.1145/3425174.3425212.