METEOR: A Tool for Monitoring Behavior Preservation in Test Code Refactorings

Tiago Samuel Rodrigues Teixeira
Institute for Technological Research
São Paulo, Brazil
tiagosamfito@gmail.com

Fábio Fagundes Silveira
Federal University of São Paulo
São José dos Campos, Brazil
fsilveira@unifesp.br

Eduardo Martins Guerra
Free University of Bolzen-Bolzano
Bozen-Bolzano, Italy
guerraem@gmail.com

ABSTRACT

The success of test refactoring, particularly in the removal of test smells, relies on a post-refactoring evaluation to ensure that the refactored test maintains its behavior, meaning its defect detection capability. Test smells are indicative of issues within test code. While various approaches in the literature propose strategies for evaluating test behavior, they lack tool support for practical industry application. Therefore, this study introduces MeteoR, a tool model that employs a mutation testing approach to assess refactored test behavior. MeteoR is implemented as a plugin for the *Eclipse* IDE and is integrated with *PITclipse*, streamlining the collection and analysis of mutation test data for effective evaluation of test refactoring. This paper provides a detailed description of the MeteoR implementation, discusses encountered challenges and findings, and includes illustrative examples demonstrating its efficacy through a preliminary evaluation, which shows promising initial results.

The video presentation of MeteoR is available at https://doi.org/10.6084/m9.figshare.25954045.

KEYWORDS

software engineering, test code refactoring, test smells, test behavior, mutation testing.

1 INTRODUCTION

Testing determines the success of refactoring application code, as emphasized by Meszaros [10]. However, regarding refactoring test code, there is no definitive approach adopted for evaluating the refactored tests, as highlighted by van Bladel and Demeyer [17].

Parsai et al. [13] and Xuan et al. [19] proposed the adoption of mutation testing to ensure that the behavior of refactored test code remains unaffected by clean-up activities, without, however, providing a tool that applies the approach they themselves proposed.

Building upon the work of Parsai et al. [13], in the previous study [16] we provided a more detailed exploration of the mutation testing-based approach. Teixeira et al. [16] introduced a test refactoring tool model, moving towards its implementation. Primarily focusing on a conceptual discussion of a holistic test refactoring tool, the study [16] covered many aspects of test refactoring and test behavior evaluation using mutation testing. However, the tool's implementation remained open for future work.

Thus, the main goal of this study is to discuss the implementation of MeteoR based on proposal approached in study [16], while also presenting three other important contributions: (1) Addressing architectural and functional aspects in the building of the tool, (2) Identifying valuable lessons learned during the development process by adopting the mutation testing approach defined in the previous studies [13, 16, 19] regarding of the test refactoring evaluation, and (3) Demonstrating the tool's suitability by applying it to two refactoring scenario examples for preliminary assessment, indicating its adequacy for adoption in test code refactoring.

The remainder of this paper is organized as follows. Section 3 reviews related work in the field of refactoring testing code providing background information on the concepts and techniques relevant to this study. Section 4 delves into the implementation details of the MeteoR. Following that, Section 5 presents the preliminary evaluation of the tool. Section 6 discusses the findings and implications of this study. The paper concludes in Section 7 with a summary of the contributions and potential future directions for research in Section 8.

Seven studies served as central references for this research [3, 7, 8, 13, 14, 17, 19]. These studies highlighted the main aspects of test code refactoring tools combined with test behavior observability approachs aiming to preserve test behavior. Additional studies were considered, and further investigation was conducted to explore additional references from databases, including ACM, Scopus (Elsevier), IEEE, and Web Of Science (WoS).

2 BACKGROUND

For a better understanding of the concepts addressed in the context of MeteoR's development, it is essential to ground the theoretical aspects of this study in two main pillars: *mutation testing* and *test refactoring*.

2.1 Mutation Testing

Mutation testing, as described by Offutt and Untch [12], objectively evaluates the adequacy of test suites through a mutation score. This score quantifies the effectiveness of the test, calculated as the proportion of "killed" mutants (those detected by the test suite) to the total number of non-equivalents. Non-equivalents are mutant versions that replicate the behavior of the original code. Mutants are faulty versions of the application used to run against the test suite.

Parsai et al. [13] and Andrade et al. [1] detail that mutation testing is carried out as follows: First, faulty versions of the application are created by intentionally injecting defects (mutations). An operator or mutator transforms a specific part of the application's code. After the faulty application versions (mutants) are generated and compiled, the test suite is executed on each of these mutants, and the results are evaluated.

Killed mutants, i.e., mutant applications that cause test failures during their execution, indicate that the testing covers the mutation and detects the faults injected. Conversely, survived mutants indicate uncovered areas, highlighting tests that need improvement. In Section 3, it is demonstrated how mutation testing fits as an approach that can be useful to assist refactoring

2.2 Test Refactoring

According to Fowler [6] in its verbal form, refactoring means restructuring the software by applying a series of refactorings in the code without altering its observable behavior. Regarding testing, refactoring is especially relevant in the context of automated tests, as it should preserve the behavior of the tests (test scenarios) without introducing errors into the test suite, as stated by Meszaros [10].

Test refactoring in this study is focused on the elimination of code smells that can lead to issues such as reduced readability, among other factors, as discussed in studies [2, 7, 9, 18]. There are various refactorings to address test smells. A foundational reference is the catalog proposed by Van Deursen et al. [18], which includes *Inline Resource Incorporation*, *Setup of External Resource*, *Making Resource Unique*, *Reducing Data*, *Adding Assertion Explanation*, and *Introducing Equality Method.*

3 RELATED WORK

Evaluation of test behavior is a crucial step in test code refactoring, aiming to detect changes in observable aspects of this behavior that may indicate a lower effectiveness of the tests. Following some approaches of test behavior observability are discussed.

3.1 Test Behavior Observability Approaches

3.1.1 Mutation Testing Approaches. Parsai et al. [13] offer a distinct perspective on mutation testing application in the test refactoring evaluation, by adopting mutation scores before and after refactoring as an indicator of change in the behavior of test. They do not merely regard mutants as a means to evaluate test quality; instead, they see them as an objective measure to gauge changes in test behavior. As related by Parsai et al. [13], any alteration in the mutation score following refactoring indicates a change in test behavior.

Xuan et al. [19] complement the proposal of Parsai et al. [13], according to their approach mutants are generated and executed against both refactored and non-refactored versions of test code. For a refactoring to be considered successful, a mutant killed by the non-refactored version of test code must also be killed by the refactored one, and a surviving mutant should persist in the refactored version of test code, indicating that, according to this perspective, the refactored test has preserved its behavior.

3.1.2 Instrumentation Approach. Pizzini [14] and [15] proposed using source code instrumentation of the *System Under Test* (SUT) and tests to identify method entry and exit points, modifications to *Class Under Test* (CUT) attributes, and loop structures. Instrumentation will allow the creation of the code execution tree, from which the behavior of tests and the SUT can be observed.

3.1.3 Static Analysis Approaches. Guerra and Fernandes [7] introduced a tool grounded in static analysis that assesses the preservation of verifications within refactored test code. A verification encompasses a single assertion and all associated actions.

In contrast, Bladel and Demeyer [3] proposed an approach based on symbolic execution. A tool called *T-CORE* (Test Code Refactoring Tool) generates a report indicating whether the test behavior has changed after refactoring. The tool captures the behavior of *Java* tests in the form of a test behavior tree using symbolic execution technique.

3.1.4 General considerations. It is worth noting that in the studies focusing on relevant approaches and tools for this research, such as [3, 7, 13–15, 19], the specific implementation of these tools was neither identified or finalized. Although some studies have addressed some aspects of the tools, they do not necessarily cover in depth the features desired, such as providing fast feedback to developers within the Integrated Development Environment (IDE). Some tools implementations have been carried out, but in some cases they are not properly available for general use or indeed have not been implemented as recognized by Parsai et al. [13].

3.2 Approach Selected

The mutation testing-based approaches [13, 19] was chosen due to its dynamic nature compared to static analysis approaches [3, 7]. Additionally, it does not require code instrumentation for behavior evaluation, unlike approaches such as those described in [14, 15]. Furthermore, the approaches [13, 19] were combined into a single approach.

Instead of solely relying on comparing mutation scores, as proposed by Parsai et al. [13], to evaluate test code behavior, it is feasible to individually compare the state of each mutant, as suggested by Xuan et al. [19] and Teixeira et al. [16]. This involves verifying whether mutants maintain their states as "survived," "killed," or "not covered" after refactoring, thereby enhancing the assurance of test behavior preservation by providing a more exhaustive analysis of the mutants' data. This approach has been incorporated into MeteoR, and the following sections will delve into its implementation specifics.

4 METEOR

MeteoR was developed as a plugin for the *Eclipse* IDE, utilizing the *Java* language. Its implementation involved the fork of the *PITclipse* project, facilitating code synergy and enhancing communication between both plugins (MeteoR ↔ *PITclipse*). *PITclipse* itself is a plugin designed to integrate *PITest* [5] with the *Eclipse* IDE.

According to Monteiro et al. [11], the *PIT or PITest*, is a widely adopted mutation testing tool for research and in the industry within the *Java* language context.

In the study [16], a test refactoring workflow was established to cover all stages (**S1** - **S9**) of a refactoring process. Each stage is detailed as follows: **S1** - *Source Code Extraction and Commit*, **S2** - *Test Smells Identification*, **S3** - *Refactoring Assistance*, **S4** (*a*, *b*) - *Mutation Test Execution and Re-execution*, **S5** - *IDE Plugin Integration*, **S6** - *Evaluation*, **S7** - *Traceability and Classification*, **S8** - *Analytics*, **S9** - *Predictive Models for Mutant Validation.*

4.1 Main MeteoR Functionalities

MeteoR was implemented considering the stages S4 (*a*,*b*), S5 and S6 from the workflow proposed by Teixeira et al. [16].

- Stage S4: Pre- and Post-Refactoring Mutation Testing:
  - S4*a*: Before applying corrections, *PITclipse* (*PITest*) is used to generate an initial listing of the state of mutants. The mutation test data serves as a baseline (*a*) for comparison after refactoring.
  - S4*b*: After refactoring, *PITclipse* is run again to execute the mutation test with the refactored tests, producing a post-refactoring listing of the state of mutants (*b*).
  - The mutation score is also collected in both executions (*a* and *b*) for comparison.
- Stage S5: Refactoring within *Eclipse* IDE:
  - Refactoring is carried out using the *Eclipse* IDE due to its extensive range of plugins, including *PITclipse* and *JUnit*.
  - Post-refactoring, tests are executed in *JUnit* to ensure 100% success against the application code without mutations.
- Stage S6: Comparison and Verification:
  - The results of the two mutation test executions (pre- and post-refactoring) are compared ($a = b$).
  - If the mutants state and mutation score remain unchanged, it indicates successful refactoring.
  - If the comparison is not positive, meaning there were any changes in mutants score or in the mutants state the developer must reassess and correct any issues in the test code before re-running the mutation test (S4*b*) and submitting changes to the GIT repository.

4.2 MeteoR Implementation Details

MeteoR acts as a trigger and handler of mutation test events executed by *PITclipse*. Indirectly, MeteoR is invoking *PITest*. Therefore, to facilitate communication between both solutions, OSGi[1] development resources from the IDE platform were utilized.

Following is presented in Figure 1 the main screen of the Eclipse IDE with MeteoR running. The UI elements are as follows: (**1**) the toolbar menu, (**2**) the main view, and (**3**) the final comparative mutation test report.

In Figure 2, the integration of the MeteoR tool with *PITclipse* is highlighted, allowing for the initiation of mutation testing through the PitLaunchShorcut class and the reception of results upon completion of mutation testing by the *PIT*.

The initiation of execution is performed through the run method of the PitMutationAgent class of the MeteoR tool. This involves invoking the launch method of the PitLaunchShortcut class, where a selectedResource object is sent as one of the arguments. This object determines the test class or package to be considered by *PIT* during mutation testing execution and is selected by the developer before starting the refactoring, through the fixation of an entry point as seen in the menu in the following section.

[1]Documentation about *OSGi* is available at: https://eclipse.dev/equinox/

4.3 MeteoR Tool Menu

In Figure 1, item (**1**) presents the menu of the MeteoR tool. The numbers from **1** to **5** seen in the menu items indicate a standard sequence of steps to be performed for each refactoring.

Following each item of menu is detailed:

(1) *Set project entry point for test:* Define the entry point for mutation testing, specifying a package or a specific test class to focus mutation testing efforts.
(2) *Create refactoring session:* Initiate a new refactoring session, recording mutation test results before and after refactoring for comparison.
(3) *Run mutation tests:* Execute mutation testing using the *PITclipse* tool, ensuring the project entry point and refactoring session are properly configured.
(4) *Set last run as baseline:* Establish the latest mutation test results as the baseline for future comparisons.
(5) *Validate refactoring:* Validate the success of refactoring by comparing mutation test results extracted before and after the code changes, generating the detailed final comparison report.

4.4 MeteoR View

The primary function of the view, item (**2**) of Figure 1, is to display current and previous refactoring session data in a tree listing. Following it is presented in details each item of the list.

Refactoring Session: Sequential number of the ongoing or completed refactoring session. Each refactoring session includes the following:

- *Baseline → Test Mutation Score:* Mutation test score set as the baseline or reference.
- *Last Result → Test Mutation Score:* Score of the last mutation test executed.
- *Refactoring Result:* After refactoring validation, the final result is presented along with the path where the detailed report, comparing mutation score and mutants state extracted before and after refactoring, is saved.

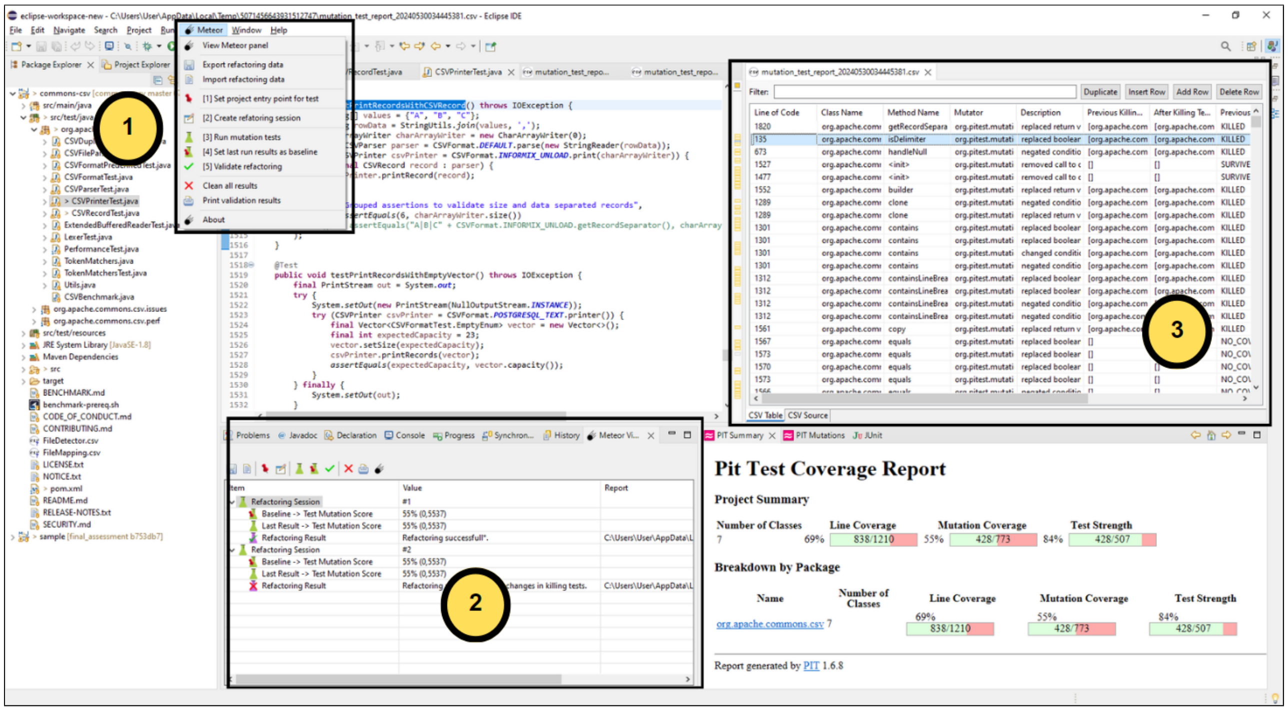
4.5 MeteoR Final Comparative Mutation Report

The item (**3**) of Figure 1 that is the final validation and comparative report of mutants state in csv format is the most important artifact handled and generated in the MeteoR tool, which can be further detailed in Table 1.

Fields that aid in identifying mutants that have undergone some state change are highlighted in red in Table 1. In case of changes in state of any mutant the field *Changed Behaviour* can be used as a filter to recover the mutants affected by the change in test behavior. Other important fields include *Previous Killing Tests* and *After Killing Tests*, which allow for the concrete identification of which test methods were involved in eliminating the mutant revealing whether a refactored test still appears on the list of killers after refactoring.

4.6 MeteoR Usage

The usage of the MeteoR tool begins with opening the *Java* project in the *Eclipse* IDE. The developer sets the entry point, representing



```
PrintRecordsWithCSVRecord() throws IOException {
g[] values = {"A", "B", "C"};
g rowData = StringUtils.join(values, ',');
rayWriter charArrayWriter = new CharArrayWriter(0);
CSVParser parser = CSVFormat.DEFAULT.parse(new StringReader(rowData));
SVPrinter csvPrinter = CSVFormat.INFORMIX_UNLOAD.print(charArrayWriter)) {
al CSVRecord record : parser) {
Printer.printRecord(record);

Grouped assertions to validate size and data separated records",
ssertEquals(6, charArrayWriter.size())
assertEquals("A|B|C" + CSVFormat.INFORMIX_UNLOAD.getRecordSeparator(), charArray
    );
}

@Test
public void testPrintRecordsWithEmptyVector() throws IOException {
    final PrintStream out = System.out;
    try {
        System.setOut(new PrintStream(NullOutputStream.INSTANCE));
        try (CSVPrinter csvPrinter = CSVFormat.POSTGRESQL_TEXT.printer()) {
            final Vector<CSVFormatTest.EmptyEnum> vector = new Vector<>();
            final int expectedCapacity = 23;
            vector.setSize(expectedCapacity);
            csvPrinter.printRecords(vector);
            assertEquals(expectedCapacity, vector.capacity());
        }
    } finally {
        System.setOut(out);
    }
```

Line of Code	Class Name	Method Name	Mutator	Description	Previous Killin...	After Killing Te...	Previous
1820	org.apache.comı	getRecordSepara	org.pitest.mutati	replaced return v	[org.apache.com	[org.apache.com	KILLED
135	org.apache.comı	isDelimiter	org.pitest.mutati	replaced boolear	[org.apache.com	[org.apache.com	KILLED
673	org.apache.comı	handleNull	org.pitest.mutati	negated conditio	[org.apache.com	[org.apache.com	KILLED
1527	org.apache.comı	<init>	org.pitest.mutati	removed call to c	[]	[]	SURVIVE
1477	org.apache.comı	<init>	org.pitest.mutati	removed call to c	[]	[]	SURVIVE
1552	org.apache.comı	builder	org.pitest.mutati	replaced return v	[org.apache.com	[org.apache.com	KILLED
1289	org.apache.comı	clone	org.pitest.mutati	negated conditio	[org.apache.com	[org.apache.com	KILLED
1289	org.apache.comı	clone	org.pitest.mutati	replaced return v	[org.apache.com	[org.apache.com	KILLED
1301	org.apache.comı	contains	org.pitest.mutati	replaced boolear	[org.apache.com	[org.apache.com	KILLED
1301	org.apache.comı	contains	org.pitest.mutati	replaced boolear	[org.apache.com	[org.apache.com	KILLED
1301	org.apache.comı	contains	org.pitest.mutati	changed conditi	[org.apache.com	[org.apache.com	KILLED
1301	org.apache.comı	contains	org.pitest.mutati	negated conditio	[org.apache.com	[org.apache.com	KILLED
1312	org.apache.comı	containsLineBrea	org.pitest.mutati	replaced boolear	[org.apache.com	[org.apache.com	KILLED
1312	org.apache.comı	containsLineBrea	org.pitest.mutati	replaced boolear	[org.apache.com	[org.apache.com	KILLED
1312	org.apache.comı	containsLineBrea	org.pitest.mutati	negated conditio	[org.apache.com	[illegible]	KILLED
1312	org.apache.comı	containsLineBrea	org.pitest.mutati	negated conditio	[org.apache.co	[illegible]	KILLED
1561	org.apache.comı	copy	org.pitest.mutati	replaced return v	[org.apache.co	[illegible]	KILLED
1567	org.apache.comı	equals	org.pitest.mutati	replaced boolear	[]	[illegible]	NO_CO\
1573	org.apache.comı	equals	org.pitest.mutati	replaced boolear	[]	[illegible]	NO_CO\
1570	org.apache.comı	equals	org.pitest.mutati	replaced boolear	[]	[]	NO_CO\
1573	org.apache.comı	equals	org.pitest.mutati	replaced boolear	[]	[]	NO_CO\

Item	Value	Report
Refactoring Session	#1	
Baseline -> Test Mutation Score	55% (0,5537)	
Last Result -> Test Mutation Score	55% (0,5537)	
Refactoring Result	Refactoring successfull*.	C:\Users\User\AppData\L
Refactoring Session	#2	
Baseline -> Test Mutation Score	55% (0,5537)	
Last Result -> Test Mutation Score	55% (0,5537)	
Refactoring Result	Refactoring [illegible] changes in killing tests.	C:\Users\User\AppData\L

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
7	69%	838/1210	55%	428/773	84%	428/507

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
org.apache.commons.csv	7	69% 838/1210	55% 428/773	84% 428/507

Report generated by PIT 1.6.8

Figure 1: IDE *Eclipse* with the MeteoR and *PITclipse* integrated.

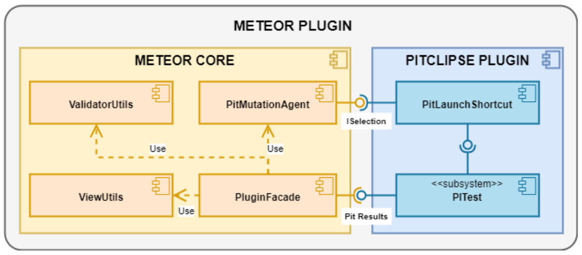


Figure 2: Component Diagram of the MeteoR.

the test class or package where the test to be refactored are located. After pining down the entry point, the developer runs the first round of mutation testing to collect data of mutants, necessary as a baseline. The tool suggests creating a refactoring session if one has not been previously created.

After completing this step, the developer proceeds to refactor the tests and conducts a second round of mutation testing. The resulting data is then compared with the baseline. If no changes are identified in the state of mutants, mutation score, or *killing tests* listings, the success of the refactoring is determined. The tool generates a detailed final comparison report in csv format for further analysis, as outlined in Section 4.5.

4.7 Adaptations on *PITest* and *PITclipse* code

It was necessary to adapt a portion of the code from *PITclipse* and *PITest* so that *PITclipse* could run in the *full mutation matrix* mode

Table 1: Description of Fields of the Final Comparative Mutation Report

Field	Description
Line of Code	Line number where the mutation was applied.
Class Name	Name of the class where the mutation was applied.
Method Name	Name of the method where the mutation was applied.
Mutator	Type of mutation applied, specified by the mutation operator.
Description	Detailed description of the mutation performed.
Previous Killing Tests	Tests that previously eliminated this mutant before refactoring.
After Killing Tests	Tests that eliminated this mutant after refactoring.
Previous Detection State	State of the mutant before refactoring (e.g., "survived," "killed," ...).
After Detection State	State of the mutant after refactoring (e.g., "survived," "killed," ...).
Changed Behaviour	Was there a change in the mutant state before and after refactoring? This field receives a boolean value that acts as a flag allowing for quick filtering of unwanted state changes, bringing up mutants that were impacted by refactoring.
Source File	Source file of the code where the mutation was applied.

provided by *PITest*[4], which had not yet been integrated into *PITclipse*. In this mode, a complete list of the *killing tests* involved in eliminating the mutant is returned. This detailed information is necessary to improve the reliability of test behavior evaluation and address the *masking effect* discussed by Parsai et al. [13]. This masking occurs when another test fails during the mutation test execution, thereby masking an error in refactoring, as will be detailed further ahead.

5 TOOL PRELIMINARY EVALUATION

To make a preliminary evaluation the tool, it was adopted a procedure as proposed by Parsai et al. [13] and Teixeira et al. [16]. Therefore, we will perform a correct refactoring and an incorrect one. It will be up to the tool to identify the correct refactoring that must not show any changes in the mutants, meaning it will not reveal any alterations in test behavior. Additionally, we will conduct an incorrect refactoring, where the tool should detect changes in some of mutants, indicating alterations in test behavior, thus flagging a problem in the refactoring. The changes are identified in the level of mutation score, mutants state or *killing tests* listings.

For this experiment, the *Apache Commons-csv* project[2] was selected, and refactorings were applied to the CSVPrinterTest class, which was then subjected to both evaluation procedures.

5.1 Correct Refactoring - Session (#1)

In the test code of the CSVPrinterTest class, particularly within the testTrimOnOneColumn method, it was observed that the assertion lacked an explanation. Consequently, following the suggestion of Van Deursen et al. [18] to include an explanation in the assertion, the refactoring was carried out. This process is illustrated in code snippets 1 and 2. Subsequently, the refactored code was validated in MeteoR, as depicted in Figure 3. As expected, MeteoR recognized the refactoring as successful.

Code Snippet 1: Test method before proper refactoring

```java
@Test
public void testTrimOnOneColumn() throws IOException {
    final StringWriter sw = new StringWriter();
    try (final CSVPrinter printer = new CSVPrinter(sw, CSVFormat.
        ↪ DEFAULT.withTrim())) {
        printer.print(" A ");
        assertEquals("A", sw.toString());
    }
}
```

Code Snippet 2: Test method after proper refactoring

```java
@Test
public void testTrimOnOneColumn() throws IOException {
    final StringWriter sw = new StringWriter();
    try (final CSVPrinter printer = new CSVPrinter(sw, CSVFormat.
        ↪ DEFAULT.withTrim())) {
        printer.print(" A ");
        assertEquals("A", sw.toString(), "Verify string A without spaces");
    }
}
```

5.2 Incorrect Refactoring - Session (#2)

It was identified in the test code of class CSVPrinterTest, specifically in the method testPrintRecordsWithCSVRecord, that the assertion could be improved. Therefore, by applying the technique of *surrounding assertions with assertAll method* as related by Martins et al. [9], the refactoring was conducted. This process is illustrated in code snippets 3 and 4. Subsequently, the refactoring was validated to check for any encountered issues. As expected, MeteoR identified the refactoring as unsuccessful.

Code Snippet 3: Test method before unproper refactoring

```java
@Test
public void testPrintRecordsWithCSVRecord() throws IOException {
    final String[] values = {"A", "B", "C"};
    final String rowData = StringUtils.join(values, ',');
    final CharArrayWriter charArrayWriter = new CharArrayWriter(0);
    try (final CSVParser parser = CSVFormat.DEFAULT.parse(new
        ↪ StringReader(rowData));
        final CSVPrinter csvPrinter = CSVFormat.INFORMIX_UNLOAD.
            ↪ print(charArrayWriter)) {
        for (final CSVRecord record : parser) {
            csvPrinter.printRecord(record);
        }
    }
    assertEquals (6, charArrayWriter.size());
    assertEquals ("A|B|C" + CSVFormat.INFORMIX_UNLOAD.
        ↪ getRecordSeparator(), charArrayWriter.toString());
}
```

As seen in line 14 of Code Snippet 4 the refactoring was performed improperly, we simulate a situation where the developer erroneously wrapped the previous assertions with assertAll. During the refactoring, he/she commented out the second assertion, affecting the test behavior.

Code Snippet 4: Test method after unproper refactoring

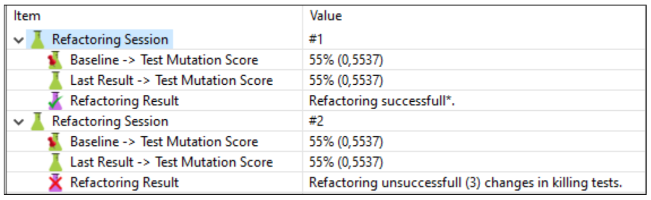
```java
1   @Test
2   public void testPrintRecordsWithCSVRecord() throws IOException {
3       final String[] values = {"A", "B", "C"};
4       final String rowData = StringUtils.join(values, ',');
5       final CharArrayWriter charArrayWriter = new CharArrayWriter(0);
6       try (final CSVParser parser = CSVFormat.DEFAULT.parse(new
            ↪ StringReader(rowData));
7           final CSVPrinter csvPrinter = CSVFormat.INFORMIX_UNLOAD.
                ↪ print(charArrayWriter)) {
8           for (final CSVRecord record : parser) {
9               csvPrinter.printRecord(record);
10          }
11      }
12      assertAll ("Grouped assertions to validate size and data separated
            ↪ records",
13          () -> assertEquals(6, charArrayWriter.size()),
14          //() -> assertEquals("A|B|C" + CSVFormat.INFORMIX_UNLOAD.
                ↪ getRecordSeparator(), charArrayWriter.toString())
15      );
16  }
```

5.3 Result Analysis

In case of succesfull refactoring (as seen in Figure 3) in the refactoring session **(#1)**, the results indicates no changes was observed in the mutation score and in the mutants state pre- and post-refactoring. In the case of refactoring session **(#2)**, the issue with the behavior change was signaled by the list of *killing tests*, as shown in Table 2.

Although the mutation score provides an objective evaluation, the MeteoR tool also assessed the state of all mutants, along with the list of *killing tests* extracted before and after refactoring.

[2]https://github.com/apache/commons-csv



Item	Value
Refactoring Session	#1
Baseline -> Test Mutation Score	55% (0,5537)
Last Result -> Test Mutation Score	55% (0,5537)
Refactoring Result	Refactoring successfull*.
Refactoring Session	#2
Baseline -> Test Mutation Score	55% (0,5537)
Last Result -> Test Mutation Score	55% (0,5537)
Refactoring Result	Refactoring unsuccessfull (3) changes in killing tests.

Figure 3: MeteoR displaying the results of evaluation of the two refactoring sessions.

During the development of the MeteoR tool, it was observed that in cases where a test improperly refactored could not be correctly validated if mutants previously killed by it were now being killed by another test that was not part of the refactoring scope. This symptom reflects the *masking effect* problem approached by Parsai et al. [13]. As *PITclipse* does not support the selection of only the relevant test methods in a more restrictive manner in the mutation test execution, it became necessary to apply these comparison in the listings of *killing tests*, in addition to validating the state of mutants as discussed by Teixeira et al. [16].

In the case of the incorrect refactoring section, despite not necessarily indicating a change in the state of mutants and in the mutation score, as demonstrated in the MeteoR view (as seen in Figure 3) in the refactoring session **(#2)**, a change was identified in the listing of *killing tests*, as seen in Table 2. MeteoR pointed out that the refactored test method no longer eliminated the mutant that was previously being eliminated. The mutants were eliminated by other tests that were not refactored.

Table 2: Comparison between *killing tests* pre- and post refactoring in one mutant state masked on incorrect refactoring

Killing Tests Before Refactoring	Killing Tests After Refactoring
testPrintRecordsWithCSVRecord	–
testParseCustomNullValues	testParseCustomNullValues
testJdbcPrinter	testJdbcPrinter
testJira135_part1	testJira135_part1
testJdbcPrinterWithResultSet	testJdbcPrinterWithResultSet
testJira135_part3	testJira135_part3
testJdbcPrinterWithResultSetHeader	testJdbcPrinterWithResultSetHeader
testJdbcPrinterWithResultSetMetaData	testJdbcPrinterWithResultSetMetaData
other test methods ...	other test methods ...

6 TOOL LIMITATIONS

Given the *masking effect* was not extensively discussed by Parsai et al. [13], we assume that when referring to the issue of masking, Parsai et al. [13] was addressing the situation where the validation result could be compromised by a different test failing on a mutant that should have been killed by the refactored test.

To address this, comparing *killing tests* data before and after refactoring is crucial. Focusing mutation tests on the refactored test methods helps prevent contamination from irrelevant tests. Despite limitations in *PITclipse* for isolated mutation testing, using *killing tests* listings comparisons proved effective.

For a thorough tool evaluation, adopting a comprehensive refactoring catalog is essential. This approach will enable a broader assessment of MeteoR's usage and limitations, revealing constraints not explored in this preliminary study focused solely on two types of test refactoring.

7 CONCLUSION

This work highlighted the nuances in the evaluation of test code refactoring applying mutation testing, emphasizing the importance of a tool designed to aid developers and avoid manual and exhaustive analysis. The primary objective, which was to implement and provide such a tool for use by both research and industrial contexts adopting mutation testing as proposed by [13, 16, 19], has been successfully accomplished. This addresses a significant gap in the availability of an integrated tool capable of assisting developers in assessing test code during the refactoring process. Prior endeavors in this area had not fully achieved this objective, as they addressed the issues without offering a finalized solution that provides fast feedback to developers within the IDE during refactoring.

The paper comprehensively covers all the essential aspects related to implementing of the MeteoR, including architectural design, integration strategy, and implementation details of the code. Additionally, lessons learned during the development phase were discussed, which involved enhancing the accuracy of previously proposed approaches. Moreover, the preliminary evaluation successfully assessed the tool's capability to identify both proper and improper refactorings.

8 FUTURE WORKS

For future work, expanding the scope of tool evaluation to include broader catalogs of refactoring and assessing it in typical production scenarios with real projects and diverse teams would be beneficial.

METEOR AND ARTIFACT AVAILABILITY

All the data related to MeteoR, such as code, evaluation data, and installation files, are available at https://github.com/meteortool. The installation instructions can be found at https://meteortool.github.io/binaries.

The MeteoR is distributed under the MIT license[3].

ACKNOWLEDGMENTS

The authors thank the São Paulo Research Foundation (FAPESP) – grant 2023/04581-0 – for financial support.

REFERENCES

[1] Stevão A. Andrade, Claudinei Brito, Misael Júnior, Ana Claudia Marciel, Gabriel Abdalla, and Márcio E. Delamaro. 2019. Analyzing the effectiveness of One-Op Mutation against the minimal set of mutants. In *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing* (Salvador, Brazil) *(SAST '19)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/3356317.3356321

[2] M. Aniche. 2022. *Effective Software Testing: A developer's guide.* Manning, NY. https://www.manning.com/books/effective-software-testing

[3] Brent van Bladel and Serge Demeyer. 2018. Test Behaviour Detection as a Test Refactoring Safety. In *Proceedings of the 2nd International Workshop on Refactoring* (Montpellier, France) *(IWoR 2018)*. Association for Computing Machinery, New York, NY, USA, 22–25. https://doi.org/10.1145/3242163.3242168

[3]Available at: https://opensource.org/license/mit

[4] Henry Coles. [n. d.]. *How to get killing tests against each mutant or a complete killing matrix with PIT?* https://stackoverflow.com/questions/74190816/how-to-get-killing-tests-against-each-mutant-or-a-complete-killing-matrix-with-p

[5] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[6] M. Fowler. 2019. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, USA.

[7] Eduardo Martins Guerra and Clovis Torres Fernandes. 2007. Refactoring Test Code Safely. In *International Conference on Software Engineering Advances (ICSEA 2007)*. 44–44. https://doi.org/10.1109/ICSEA.2007.57

[8] Rogério Marinke, Eduardo Martins Guerra, Fábio Fagundes Silveira, Rafael Monico Azevedo, Wagner Nascimento, Rodrigo Simões de Almeida, Bruno Rodrigues Demboscki, and Tiago Silva da Silva. 2019. Towards an Extensible Architecture for Refactoring Test Code. In *Computational Science and Its Applications – ICCSA 2019*, Sanjay Misra, Osvaldo Gervasi, Beniamino Murgante, Elena Stankova, Vladimir Korkhov, Carmelo Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, and Eufemia Tarantino (Eds.). Springer International Publishing, Cham, 456–471.

[9] Luana Martins, Taher Ghaleb, Heitor Costa, and Ivan Machado. 2024. A comprehensive catalog of refactoring strategies to handle test smells in Java-based systems. *Software Quality Journal* (03 2024), 1–39. https://doi.org/10.1007/s11219-024-09663-7

[10] Gerard Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code.* Pearson Education.

[11] Ricardo Monteiro, Vinicius Humberto Serapilha Durelli, Marcelo Eler, and Andre Endo. 2022. An Empirical Analysis of Two Mutation Testing Tools for Java. In *Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing* (Uberlandia, Brazil) *(SAST '22)*. Association for Computing Machinery, New York, NY, USA, 49–58. https://doi.org/10.1145/3559744.3559751

[12] A. Jefferson Offutt and Roland H. Untch. 2001. *Mutation 2000: Uniting the Orthogonal.* Springer US, Boston, MA. 34–44 pages. https://doi.org/10.1007/978-1-4757-5939-6_7

[13] Ali Parsai, Alessandro Murgia, Quinten David Soetens, and Serge Demeyer. 2015. Mutation Testing as a Safety Net for Test Code Refactoring. In *Scientific Workshop Proceedings of the XP2015* (Helsinki, Finland) *(XP '15 workshops)*. Association for Computing Machinery, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2764979.2764987

[14] Adriano Pizzini. 2022. Behavior-based test smells refactoring : Toward an automatic approach to refactoring Eager Test and Lazy Test smells. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 261–263. https://doi.org/10.1145/3510454.3517059

[15] Adriano Pizzini, Sheila Reinehr, and Andreia Malucelli. 2023. Sentinel: A process for automatic removing of Test Smells. In *Proceedings of the XXII Brazilian Symposium on Software Quality* (, Brasília, Brazil,) *(SBQS '23)*. Association for Computing Machinery, New York, NY, USA, 80–89. https://doi.org/10.1145/3629479.3630019

[16] Tiago Samuel Rodrigues Teixeira, Fábio Fagundes Silveira, and Eduardo Martins Guerra. 2023. Moving towards a Mutant-Based Testing Tool for Verifying Behavior Maintenance in Test Code Refactorings. *Computers* 12, 11 (2023). https://doi.org/10.3390/computers12110230

[17] Brent van Bladel and Serge Demeyer. 2017. Test Refactoring: a Research Agenda. In *Proceedings SATToSE*.

[18] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, M. Marchesi (Ed.).

[19] Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, and Martin Monperrus. 2016. B-Refactoring: Automatic test code refactoring to improve dynamic analysis. *Information and Software Technology* 76 (2016), 65–80. https://doi.org/10.1016/j.infsof.2016.04.016