# How to Kill Them All: An Exploratory Study on the Impact of Code Observability on Mutation Testing

**Qianqian Zhu, Andy Zaidman, and Annibale Panichella**

**Software Engineering Research Group, Delft University of Technology, Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands**

Corresponding author:
Qianqian Zhu

Email address: qianqian.zhu@tudelft.nl

## ABSTRACT

Mutation testing has been well-known for its efficacy to assess test quality, and recently it has started to be applied in industry as well. However, what should a developer do when confronted with a low mutation score? Should the test suite be reinforced to increase the mutation score, or should the production code be improved as well, to make the creation of better tests possible? In this paper, we investigate whether testability and observability metrics are correlated with the mutation score on six open source Java projects. We observe a correlation between observability metrics and the mutation score, e.g., test directness seems to be an essential factor. Based on our insights from the correlation study, we propose a number of "mutation score anti-patterns", which enable software engineers to refactor their existing code to be able to improve the mutation score. In doing so, we observe that relatively simple refactoring operations enable an improvement in the mutation score.

## 1 INTRODUCTION

Mutation testing has been a very active research field since the 1970s as a technique to evaluate the quality of test suites [1]. Recent advances have made it possible for mutation testing to be used in industry [2]. For example, *PIT/PiTest* [3] has been adopted by several companies, such as *The Ladders* and *British Sky Broadcasting* [4]. Furthermore, *Google* [5] has integrated mutation testing with the code review process for around 6000 software engineers.

As mutation testing gains traction in industry, a better understanding of the *mutation score* (the outcome of mutation testing) becomes essential. This is because once mutation testing is adopted, project managers would consider the mutation score as an important metric to monitor testing activities. However, in our previous study, we have established that certain mutants could be killed only after refactoring the production code to increase the observability of state changes [6]. This leads us to our hypothesis that *code quality* plays an essential role in mutation testing. More specifically, we conjecture that *software testability* and *code observability* are two key factors. *Testability* is defined as the "attributes of software that bear on the effort needed to validate the software product" [7, 8]. Given our context, an important part of testability is *observability*, which is a measure of how well internal states of a system can be inferred, usually through the values of its external outputs [9]. More specifically, observability indicates how a failure that is triggered propagates through the code and becomes observable to either the tester or an automated comparator [10].

Inspired by the work of Bruntink and van Deursen [8], who have explored the relation between nine object-oriented metrics and testability, we investigate the relationhip between code quality metrics and mutation testing. Our first three research questions steer our investigation:

**RQ1** What is the relation between *testability* metrics and the mutation score?

**RQ2** What is the relation between *observability* metrics and the mutation score?

**RQ3** What is the relation between the combination of *testability* and *observability* metrics and the mutation score?

After investigating the relationship between *testability*, *observability*, and mutation testing, we still lack insight into how these relationships can be made actionable for software engineers in practice. That is why, based on the observations from **RQ1-RQ3**, we define anti-patterns or indicators that software engineers can apply to their code to ensure that mutants can be killed. This leads us to the next research question:

**RQ4** To what extent does the refactoring of anti-patterns based on *testability* and *observability* help in improving the mutation score?

In terms of the methodology that we follow in our study, for **RQ1-RQ3**, we use statistical analysis on open-source Java projects to investigate the relationship between testability, observability, and the mutation score. For **RQ4**, we perform a case study involving 16 code fragments to investigate whether the refactoring of anti-patterns improves the mutation score.

## 2 BACKGROUND

In this section, we briefly introduce basic concepts and related work on mutation testing and testability metrics.

### 2.1 Mutation Testing

Mutation testing is defined by Jia and Harman [1] as a fault-based testing technique which provides a testing criterion called the *mutation adequacy score*. This score can be used to measure the effectiveness of a test suite regarding its ability to detect faults [1]. The principle of mutation testing is to introduce syntactic changes into the original program to generate faulty versions (called *mutants*) according to well-defined rules (mutation operators) [11]. The benefits of mutation testing have been extensively investigated and can be summarised as [12]: 1) having better fault exposing capability compared to other test coverage criteria [13, 14, 15], 2) being an excellent alternative to real faults and providing a good indication of the fault detection ability of a test suite [16, 17].

Researchers have actively investigated mutation testing for decades (evidenced by the extensive survey [11, 1, 18, 12]). Recently, it has started to attract attention from industry [2]. In part, this is due to the growing awareness of the importance of testing in software development [19]. *Code coverage*, the most common metric to measure test suite effectiveness, has seen its limitations being reported in numerous studies (e.g. [13, 14, 15, 20]). Using structural coverage metrics alone might be misleading because in many cases statements might be covered, but their consequences might not be asserted [20]. Another factor is that a number of well-developed open-source mutation testing tools (e.g., PIT/PiTest [3] and Mull [21]) have contributed to mutation testing being applied in industrial environments [2, 5, 4].

However, questions still exist about mutation testing, especially regarding the usefulness of a mutant [22]. The majority of the mutants generated by existing mutation operators are equivalent, trivial and redundant [23, 22, 24, 25, 26], which reduces the efficacy of the mutation score. If a class has a high mutation score while most mutants generated are trivial and redundant, the high mutation score does not promise high test effectiveness. A better understanding of mutation score and mutants is thus urgent.

To address this knowledge gap, numerous studies have investigated how useful mutants are. Example studies include *mutant subsumption* [23], *stubborn* mutants [27], and *real-fault coupling* [17, 25]. These studies paid attention to the context and types of mutants as well as the impact of the test suite, while the impact of production code quality has rarely been investigated. We have seen how code quality can influence how hard it is to test [8] (called software testability [28]), and since mutation testing can generally be considered as "testing the tests," production code quality could also impact mutation testing. Due to the lack of insights into how code quality affects the efforts needed for mutation testing, we conduct this exploratory study. Our study can help researchers and practitioners deepen their understanding of the mutation score, which is generally related to test suite quality and mutant usefulness.

### 2.2 Existing Object-Oriented Metrics for Testability

The notion of *software testability* dates back to 1991 when Freedman [28] formally defined *observability* and *controllability* in software domain. Voas [29] proposed a dynamic technique coined propagation,

**Table 1.** Summary of Method-Level Code Quality Metrics

| Abbr. | Full name | Description |
|---|---|---|
| COMP | Cyclomatic Complexity | McCabes cyclomatic Complexity for the method |
| NOA | Number of Arguments | The number of Arguments |
| NOCL | Number of Comments | The number of Comments associated with the method |
| NOC | Number of Comment Lines | The number of Comment Lines associated with the method |
| VDEC | Variable Declarations | The number of variables declared in the method |
| VREF | Variable References | The number of variables referenced in the method |
| NOS | Number of Java statements | The number of statements in the method |
| NEXP | Number of expressions | The number of expressions in the method |
| MDN | Max depth of nesting | The maximum depth of nesting in the method |
| HLTH | Halstead length | The Halstead length of the metric (one of the Halstead Metrics) |
| HVOC | Halstead vocabulary | The Halstead vocabulary of the method (one of the Halstead Metrics) |
| HVOL | Halstead volume | The Halstead volume of the method (one of the Halstead Metrics) |
| HDIF | Halstead difficulty | The Halstead difficulty of the method (one of the Halstead Metrics) |
| HEFF | Halstead effort | The Halstead effort of the method (one of the Halstead Metrics) |
| HBUG | Halstead bugs | The Halstead prediction of the number of bugs in the method (one of the Halstead Metrics) |
| TDN | Total depth of nesting | The total depth of nesting in the method |
| CAST | Number of casts | The number of class casts in the method |
| LOOP | Number of loops | The number of loops (for, while) in the method |
| NOPR | Number of operators | The total number of operators in the method |
| NAND | Number of operands | The total number of operands in the method |
| CREF | Number of classes referenced | The classes referenced in the method |
| XMET | Number of external methods | The external methods called by the method |
| LMET | Number of local methods | The number of methods local to this class called by this method |
| EXCR | Number of exceptions referenced | The number of exceptions referenced by the method |
| EXCT | Number of exceptions thrown | The number of exceptions thrown by the method |
| MOD | Number of modifiers | The number of modifiers (public, protected, etc.) in method declaration |
| NLOC | Lines of Code | The number of lines of code in the method |

infection and execution (PIE) analysis for statistically estimating the program's *fault sensitivity*. More recently, researchers have aimed to make further understandings of *testability* by using statistical methods to predict *testability* based on various code metrics. Influential works include Bruntink and van Deursen [8] that they explored the relationship between nine object-oriented metrics and testability. To explore the relation between *testability* and mutation score (**RQ1**), we first need collect a number of existing object-oriented metrics which have been proposed in the literature. In total, we collect 64 code quality metrics, including both class-level and method-level metrics that have been the most widely used.

We present a brief summary of the 64 metrics in Table 1 (method-level) and Tables 2–3 (class-level). These metrics have been computed using a static code analysis tool provided by `JHawk` [30].

## 3 MUTANT OBSERVABILITY

To explore the relation between *observability* and mutation score (**RQ2**), we first need a set of metrics to quantify *mutant observability*. The concept of *observability* originates from dynamical systems and automata [31]. Whalen et al. [32] formally defined *observability* as follows: an expression in a program is *observable* in a *test case* if the value of an expression is changed, leaving the rest of the program intact, and the output of the system is changed correspondingly. If there is no such a value, then the expression is not *observable* for that test.

According to Whalen et al. [32]'s definition of observability, we consider that *mutant observability* comprises two perspectives: that of production code and that of the test case. To better explain these two perspectives, let us consider the example in Listing 1 from project `jfreechart-1.5.0` and its corresponding test. This method is to set the line paint for `LegendItem` object. There is one mutant in Line 2 that removes call to `org.jfree.chart.util.Args::null-NotPermitted`. This mutant is not killed by `testSerialization`. Looking at the *observability* of this mutant from the *production code* perspective, we can see that `setLinePaint` method is void; thus, this mutant is hard to detect because there is no return value for the test case to *assert*. From the *test case* perspective, although `testSerialization` invokes method `setLinePaint` in Line 12, no proper assertion statements are used

**Table 2.** Summary of Class-Level Code Quality Metrics (1)

| Abbr. | Full name | Description |
|---|---|---|
| NOMT | Number of methods | The number of methods in the class (WMC - one of the Chidamber and Kemerer metrics) |
| LCOM | Lack of Cohesion of Methods | The value of the Lack of Cohesion of Methods metric for the class. This uses the LCOM* (or LCOM5) calculation. (one of the Chidamber and Kemerer metrics) |
| TCC | Total Cyclomatic Complexity | The total McCabes cyclomatic Complexity for the class |
| AVCC | Average Cyclomatic Complexity | The average McCabes cyclomatic Complexity for all of the methods in the class |
| MAXCC | Maximum Cyclomatic Complexity | The maximum McCabes cyclomatic Complexity for all of the methods in the class |
| NOS | Number of Java statements | The number of statements in the class |
| HLTH | Cumulative Halstead length | The Halstead length of the code in the class plus the total of all the Halstead lengths of all the methods in the class |
| HVOL | Cumulative Halstead volume | The Halstead volume of the code in the class plus the total of all the Halstead volumes of all the methods in the class |
| HEFF | Cumulative Halstead effort | The Halstead effort of the code in the class plus the total of all the Halstead efforts of all the methods in the class |
| HBUG | Cumulative Halstead bugs | The Halstead prediction of the number of bugs in the code of the class and all of its methods |
| UWCS | Un Weighted class Size | The Unweighted Class Size of the class |
| NQU | Number of Queries | The number of methods in the class that are queries (i.e. that return a value) |
| NCO | Number of Commands | The number of methods in the class that are commands (i.e. that do not return a value) |
| EXT | External method calls | The number of external methods called by the class and by methods in the class |
| LMC | Local method calls | The number of methods called by the class and by methods in the class |
| HIER | Hierarchy method calls | The number of local methods called by the class and by methods in the class that are defined in the hierarchy of the class |
| INST | Instance Variables | The number of instance variables declared in the class |
| MOD | Number of Modifiers | The number of modifiers (public, protected etc) applied to the declaration of the class |
| INTR | Number of Interfaces | The number of interfaces implemented by the class |

**Table 3.** Summary of Class-Level Code Quality Metrics (2)

| Abbr. | Full name | Description |
|---|---|---|
| PACK | Number of Packages imported | The number of packages imported by the class |
| RFC | Response for Class | The value of the Response For Class metric for the class. (One of the Chidamber and Kemerer metrics) |
| MPC | Message passing | The value of the Message passing metric for the class |
| CBO | Coupling between objects | The value of the Coupling Between Objects metric for the class. (One of the Chidamber and Kemerer metrics) |
| FIN | Fan In | The value of the Fan In (Afferent coupling (Ca)) metric for the class |
| FOUT | Fan Out | The value of the Fan Out (Efferent coupling (Ce)) metric for the class |
| R-R | Reuse Ratio | The value of the Reuse Ratio for the class |
| S-R | Specialization Ratio | The value of the Specialization Ratio for the class |
| NSUP | Number of Superclasses | The number of superclasses (excluding Object) in the hierarchy of the class |
| NSUB | Number of Subclasses | The number of subclasses below the class in the hierarchy. (NOC - one of the Chidamber and Kemerer metrics) |
| MI | Maintainability Index (including comments) | The Maintainability Index for the class, including the adjustment for comments |
| MINC | Maintainability Index (not including comments) | The Maintainability Index for the class without any adjustment for comments |
| COH | Cohesion | The value of the Cohesion metric for the class |
| DIT | Depth of Inheritance Tree | The value of the Depth of Inheritance Tree metric for the class. (One of the Chidamber and Kemerer metrics) |
| LCOM2 | Lack of Cohesion of Methods (variant 2) | The value of the Lack of Cohesion of Methods (2) metric for the class.This uses the LCOM2 calculation. (One of the Chidamber and Kemerer metrics) |
| CCOM | Number of Comments | The number of Comments associated with the class |
| CCML | Number of Comment Lines | The number of Comment Lines associated with the class |
| NLOC | Lines of Code | The number of lines of code in the class and its methods |

to examine the changes of `Args.nullNotPermitted()` which is used to check that the object `paint` is not null.

Starting with two angles of mutant observability, we come up with a set of the mutant observability metrics. First of all, the return type of the method. As discussed in Listing 1, in a void method is hard

```java
public void setLinePaint(Paint paint) {
    Args.nullNotPermitted(paint, "paint");
    this.linePaint = paint;
}

@Test
public void testSerialization() {
    LegendItem item1 = new LegendItem("Item", "Description",
            "ToolTip", "URL",
            new Rectangle2D.Double(1.0, 2.0, 3.0, 4.0), new GradientPaint(
                    5.0f, 6.0f, Color.BLUE, 7.0f, 8.0f, Color.GRAY));
    item1.setLinePaint(new GradientPaint(1.0f, 2.0f, Color.WHITE, 3.0f,
            4.0f, Color.RED));
    LegendItem item2;
    item2 = (LegendItem) TestUtils.serialised(item1);
    assertEquals(item1, item2);
}
```

**Listing 1.** Example of method `org.jfree.chart.LegendItem:setLinePaint` and its test

```java
    private static String getMantissa(final String str) {
        return getMantissa(str, str.length());
    }
```

**Listing 2.** Example of method `getMantissa` in class `NumberUtils`

to observe the changing states inside the method because there is no return value for test cases to assert. Accordingly, we design two metrics, `is_void` and `non_void_percent` (shown in 1st and 2nd rows in Table 5). Besides these two, a void method mostly changes the field(s) of the class it belongs to. A workaround to test a *void* method is to invoke getters. So `getter_percentage` (shown in 3rd row in Table 5) is proposed to complements `is_void`.

Second, the access control modifiers. Let us consider the example in Listing 2. The method `getMantissa` in class `NumberUtils` returns the mantissa of the given number. This method has only one mutant: the return value is replaced with "`if (x != null) null else throw new RuntimeException`". This mutant should be easy to detect given an input of either a legal number (the return value cannot be null) or a null string (throw an exception). The reason this "trivial" mutant is not detected is because the method `getMantissa` is private. The access control modifier *private* makes it impossible to *directly* test the method `getMantissa`, for this method is only visible to methods from class `NumberUtils`. To test this method, the test case must first invoke a method that calls method `getMantissa`. From this case, we observe that access control modifiers influence the visibility of the method, so as to play a significant role in mutant observability. Thereby, we take access control modifiers into account to quantify mutant observability, where we design `is_public` and `is_static` (shown in 4th and 5th rows in Table 5).

Third, fault masking. We observe that mutants generated in certain locations are more likely to be *masked* [33], i.e., the state change cannot propagate to the output of the method. The first observation is the mutants in a nested class, thus we come up with `is_nested` (in 6th row in Table 5). The next case is the mutants generated inside nested conditions and loops, where we define `nested_depth` (shown in 7th row in Table 5) and a set of metrics to quantify the conditions and loops (shown in 8th - 13rd rows in Table 5). The last observation is the mutants in a long method, thus we design `method_length` (shown in 14th row in Table 5).

Next, test directness. For instance, Listing 3 shows the class `Triple`, which is an abstract implementation defining the basic functions of the object and that consists of three elements. It refers to the elements as "left", "middle" and "right". The method `hashCode` returns the hash code of the object. Six mutants are generated for the method `hashCode` in class `Triple`. Table 4 summarises all the mutants from Listing 3. Of those six mutants, only Mutant 1 is killed, and the other mutants are not equivalent. Through further investigation of method `hashCode` and its test class, we found that although this method has 100% coverage by the test suite, there is no *direct* test for this method. A *direct* test would mean that the method is directly invoked by the test methods [34]. The direct test is useful because it allows to

```
@Override
public int hashCode() {
    return (getLeft() == null ? 0 : getLeft().hashCode()) ^
        (getMiddle() == null ? 0 : getMiddle().hashCode()) ^
        (getRight() == null ? 0 : getRight().hashCode());
}
```

**Listing 3.** Example of method hashCode in class Triple

**Table 4.** Summary of mutants from Listing 3

| ID | Line No. | Mutator | Results |
|---|---|---|---|
| 1 | 3 | negated conditional | Killed |
| 2 | 3 | replaced return of integer sized value with (x == 0 ? 1 : 0) | Survived |
| 3 | 3 | Replaced XOR with AND | Survived |
| 4 | 4 | negated conditional | Survived |
| 5 | 4 | Replaced XOR with AND | Survived |
| 6 | 5 | negated conditional | Survived |

directly control the input data plus to directly assert the output of a method. This example shows that test directness has a considerable impact on mutation testing, which denotes the test case angle of *mutant observability*. Therefore, we design two metrics, `direct_test_no.` and `test_distance` (shown in 15th and 16th row in Table 5), to quantify test directness. These two metrics represent the test case perspective of *mutant observability*.

The last but not the least, the assertion. As discussed in Listing 1, we observe that mutants without appropriate assertions in place cannot be killed, as a prerequisite to killing a mutant is to have the tests fail in the mutated program. Accordingly, we come up with three metrics to quantify assertions in the method, `assertion_no.`, `assertion-McCabe_Ratio` and `assertion_density` (shown in 17th - 19th rows in Table 5). These three metrics are proposed based on the test case perspective of *mutant observability*.

To sum up, Table 5 presents all the mutant observability metrics we propose, where we display the name, the definition of each metric and the category.

## 4 EXPERIMENTAL SETUP

To examine our conjectures, we conduct an experiment using six open-source projects. We sum up all the research questions we have proposed in Section 1:

- **RQ1**: *What is the relation between* testability *metrics and the mutation score?*
- **RQ2:** *What is the relation between* observability *metrics and the mutation score?*
- **RQ3:** *What is the relation between the combination of* testability *and* observability *metrics and the mutation score?*
- **RQ4:** *To what extent does refactoring of anti-patterns based on testability and observability help in improving the mutation score?*

### 4.1 Subject Systems

We use six systems publicly available on GitHub in this experiment. Table 6 summarises the main characteristics of the selected projects. These systems are selected because they have been widely used in the research domain [35]. All systems are written in Java, and tested by means of JUnit. The granularity of our analysis is at method-level.

The results of the mutants that are killable for all of the subjects are shown in Columns 7-8 of Table 6. Figure 1a shows the distribution of mutation score among all methods. The majority of the mutation scores are either 0 or 1. Together with Figure 1b, we can see that the massive number of 0 and 1 are due to the low mutant number per method. Most methods only have less than 5 mutants.

**Table 5.** Summary of mutant observability metrics

| # | Name | Definition | Category |
|---|---|---|---|
| 1 | is_void | whether the return value of the method is void or not | return type |
| 2 | non_void_percent (class-level) | the percent of non-void methods in the class | |
| 3 | getter_percentage | the percentage of getter methods in the class[1] | |
| 4 | is_public | whether the method is public or not | access control modifiers |
| 5 | is_static | whether the method is static or not | |
| 6 | is_nested (class-level) | whether the method is located in a nested class or not | fault masking |
| 7 | nested_depth | the maximum number of nested depth (`MDN` from Section 2.2 | |
| 8 | (cond) | the number of conditions (`if`, `if-else` and `switch`) in the method | |
| 9 | (cond(cond)) | the number of nested conditions (e.g. `if{if{}}`) in the method | |
| 10 | (cond(loop)) | the number of nested condition-loops (e.g. `if{for{}}`) in the method | |
| 11 | (loop) | the number of loops (`for`, `while` and `do-while`) in the method (`LOOP` from Section 2.2) | |
| 12 | (loop(cond)) | the number of nested loop-conditions (e.g. `for{if{}}`) in the method. | |
| 13 | (loop(loop)) | the number of nested loop-conditions (e.g. `for{for{}}`) in the method. | |
| 14 | method_length | the number of lines of code in the method (`NLOC` from Section 2.2) | |
| 15 | direct_test_no. | the number of methods directly invoked by the test methods[2] | test directness |
| 16 | test_distance | the shortest method call sequence required to invoke the method in test methods[3] | |
| 17 | assertion_no. | the number of assertions in direct tests | assertion |
| 18 | assertion-McCabe_Ratio | the ratio between the total number of assertions in direct tests and the McCabe Cyclomatic complexity | |
| 19 | assertion_density | the ratio between the total number of assertions in direct tests and the lines of code in direct tests | |

[1]A getter method must follow three patterns [39]: (1) must be public; (2) has no arguments and its return type must be something other than void. (3) have naming conventions: the name of a getter method begins with "get" followed by an uppercase letter.
[2]If the method is not directly tested, then the direct test no. is 0.
[3]If the method is directly tested, then the distance is 0. The maximum distance is set `Integer.MAX_VALUE` in Java which means there is no method call sequence that can reach the test methods.

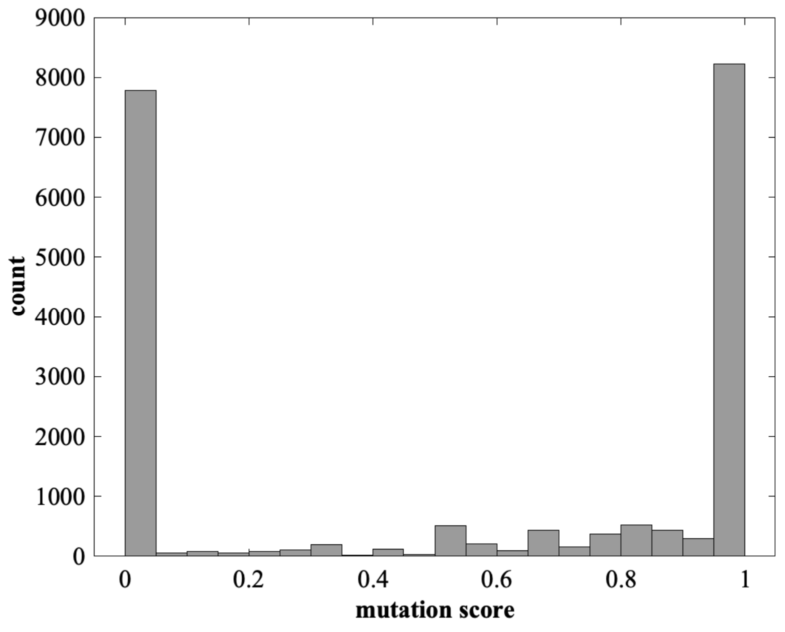
**Table 6.** Subject Systems

| PID | Project | LOC | #Tests | #Methods | | #Mutants | |
|---|---|---|---|---|---|---|---|
| | | | | #Total | #Selected | #Total | #Killed |
| 1 | Bukkit-1.7.9-R0.2 | 32373 | 432 | 7325 | 2385 | 7325 | 947 |
| 2 | commons-lang-LANG_3_7 | 77224 | 4068 | 13052 | 2740 | 13052 | 11284 |
| 3 | commons-math-MATH_3_6_1 | 208959 | 6523 | 48524 | 6663 | 48524 | 38016 |
| 4 | java-apns-apns-0.2.3 | 3418 | 91 | 429 | 150 | 429 | 247 |
| 5 | jfreechart-1.5.0 | 134117 | 2175 | 34488 | 7133 | 34488 | 11527 |
| 6 | pysonar2-2.1 | 10926 | 269 | 3070 | 719 | 3074 | 836 |
| | Overall | 467017 | 13558 | 106888 | 19790 | 106892 | 62857 |

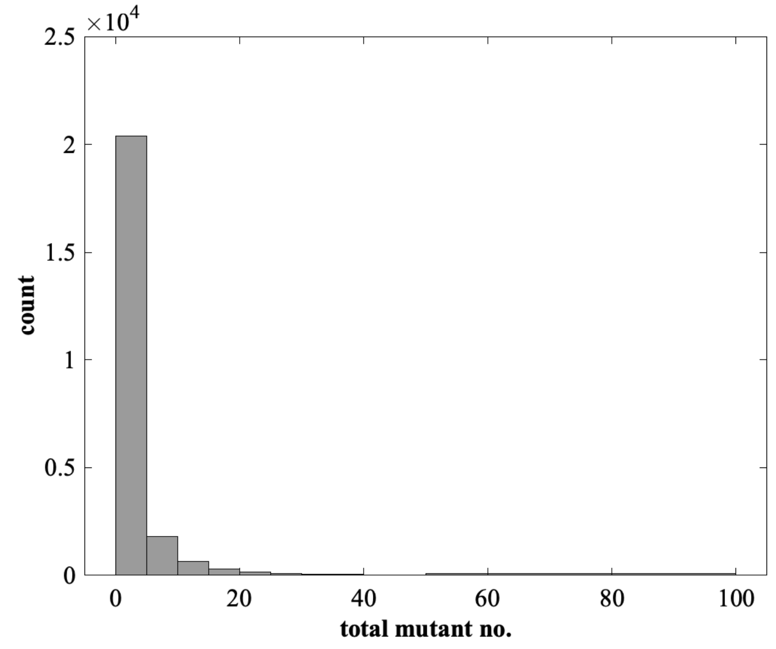
## 4.2 Tool implementation

To evaluate the *mutant observability metrics* that we have proposed, we implement a prototype tool (coined MUTATION OBSERVER) to capture all the necessary information from both the program under test and the mutation testing process. This tool is openly available on GitHub [36].

Our tool extracts information from three parts of the system under test (in Java): source code, bytecode, and tests. Firstly, Antlr [37] parses the source code to obtain the basic code features, e.g., **is public**, **is static** and **(cond)**. Secondly, we adopt Apache Commons BCEL [38] to parse the bytecode. Then, `java-callgraph` [39] generates the pairs of method calls between the source code and tests, which we later use to calculate **direct test no.** and other test call related metrics. The last part is related to the mutation testing process, for which we adopt PiTest [3] to obtain the killable mutant results. An overview



**(a)** Distribution of mutation score per method



**(b)** Distribution of total mutant no. per method

of the architecture of MUTATION OBSERVER can be seen in Figure 2.

The mutation operators we adopt are the *default* mutation operators provided by PitTest [40]: `Conditionals Boundary Mutator`, `Increments Mutator`, `Invert Negatives Mutator`, `Math Mutator`, `Negate Conditionals Mutator`, `Return Values Mutator` and `Void Method Calls Mutator`.

## 4.3 Design of Experiment

To answer **RQ1** and **RQ2**, we first adopt Spearman's rank-order correlation to statistically measure the correlation between each metric (both existing code metrics and mutant observability metrics) and the mutation score of the corresponding methods or classes. Spearman's correlation test checks whether there

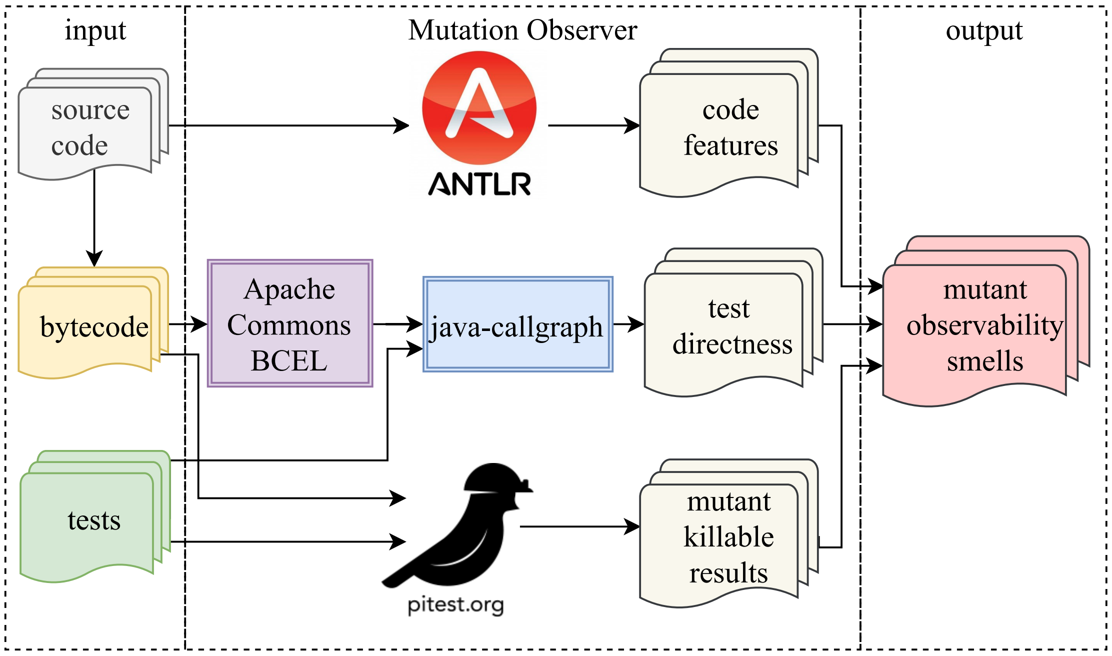


**Figure 2.** Overview of MUTATION OBSERVER architecture

exists a monotonic relationship (linear or not) between two data samples. It is a *non-parametric* test and, therefore, it does not make any assumption about the distribution of the data being tested. The resulting coefficient $\rho$ takes values in the interval $[-1;+1]$; the higher the correlation in either direction (positive or negative), the stronger the monotonic relationship between the two data samples under analysis. The strength of the correlation can be established by classifying the into "negligible" ($|\rho| < 0.1$), "small" ($0.1 \leq |\rho| < 0.3$), "medium" ($0.3 \leq |\rho| < 0.5$), and "large" ($|\rho| \geq 0.5$) [41]. Positive $\rho$ values indicate that one distribution increases when the other increases as well; negative $\rho$ values indicate that one distribution decreases when the other increases.

The mutation score is calculated by Equation 1 (method-level).

$$mutation\,score\,(A) = \frac{\#\,killed\,mutants\,in\,method\,A}{\#\,total\,mutants\,in\,method\,A}1 \tag{1}$$

To calculate Spearman's rank-order correlation coefficient between each metric and mutation score, we adopt Matlab to conduct the statistical analysis (`corr` function with the option of "`Spearman`" in Matlab's default package).

Moreover, except for the pair-wise correlations between each metric and mutation score, we are also interested in how those metrics interact with each other. To do so, we first turn the correlation problem into a binary classification problem. We use 0.5 as the cutoff between `HIGH` and `LOW` mutation core. We consider all the metrics to predicat whether the method belongs to classes with `HIGH` or `LOW` mutation score. For prediction, we adopt *Random Forest* [42] as the classification algorithm, where we use `WEKA` [43] to build the prediction model. Random Forest is an ensemble method based on a collection of decision tree classifiers , where the individual decision trees are generated using a random selection of attributes at each node to determine the split [44]. Thus, random forest is more accurate than one decision tree, and overfitting is not a problem [44]. In terms of feature importance, we apply `scikit-learn` [45] to conduct the analysis.

To answer **RQ3**, we first compare the results of Spearman's Rank-Order Correlation analysis between existing code metrics and mutant observability metrics in terms of *rho* and *p-value*. Then, we compare the results of the prediction models generated by the *Random Forests* learning method. More specifically, for each project, we compare three types of classification models: (1) a model based on merely existing code metrics, (2) a model based on merely mutant observability metrics, and (3) a model based on the combination of existing and our observability metrics (overlapping metrics, e.g., `method_length` to `NLOC`, are only considered once). To examine the effectiveness of random forests in our dataset, we also consider ZeroR, which classifies all the instances to the majority and ignores all predictors, as the baseline. It might be that our data is not balanced, as in that one project has over 90% methods with a HIGH mutation score. This could entail that the classification model achieving 90% accuracy is not necessarily an effective model. In this situation, ZeroR could also achieve over 90% accuracy in that scenario. Our random forests model must thus perform better than ZeroR; otherwise, the random forests model is not suitable for our dataset.

In total, we consider four classification models: 1) ZeroR, 2) random forests model based on existing metrics, 3) random forests model based on mutant observability metrics, and 4) random forests model based on the combination of existing metrics and mutant observability metrics.

To answer **RQ4**, we first need to establish the anti-patterns (or smells) based on these metrics. An example of an anti-pattern rule generated from the metrics is: `method_length` $> 20$ and `test_distance` $> 2$. In this case, it is highly likely that the method has low mutation score. To obtain the anti-pattern rules, we adopt J48 to build a decision tree [46, 43]. We consider J48 because of its advantage in interpretation over random forests. After building the decision tree, we rank all leaf (or paths) according to instances falling into each leaf and accuracy. We select the leaves with the highest instances and accuracy $\geq 0.8$ for further manual analysis, to understand to what extent refactoring of the anti-patterns can help in improving the mutation score.

### 4.4 Evaluation Metrics

For **RQ1**, **RQ2**, and **RQ3**, to ease the comparisons of the four classification models, we consider four metrics which have been widely used in classification problems: precision, recall, AUC, and the mean absolute error. To that end, we first introduce four key notations: TP, FP, FN, and TN, which denotes true positive, false positive, false negative, and true negative, respectively.

**Precision** The fraction of true positive instances in the instances that are predicted to be positive: TP/(TP+FP). The higher the precision is, the fewer false positive errors there are.

**Recall** The fraction of true positive instances in the instances that are actual positives: TP/(TP+FN). The higher the recall, the fewer false negative errors there are.

**AUC** The area under ROC curve, which measures the overall discrimination ability of a classifier. An area of 1 represents a perfect test; an area of 0.5 represents a worthless test.

**Mean absolute error** The mean of overall differences between the predicted values and actual values.

## 5 RQ1 - RQ3 TESTABILITY VERSUS OBSERVABILITY VERSUS COMBINATION

We opt to discuss the three research questions, **RQ1**, **RQ2** and **RQ3**, together, because it gives us the opportunity to compare testability, observability and their combination in detail.

### 5.1 Spearman's rank order correlation

#### *5.1.1 Testability*

**Findings** Table 7 presents the overall results of Spearman's rank-order correlation analysis for existing code metrics. The columns of "rho" represent the pairwise correlation coefficient between each code metrics and the mutation score. The *p-values* columns denote the strength of evidence for testing the hypothesis of no correlation against the alternative hypothesis of a non-zero correlation using Spearman's rank-order. Here we used 0.05 as the cut-off for significance. From Table 7 , we can see that except for NOS, NLOC, MOD, EXCR, INST(class), NSUB(class), COH(class) and S-R(class) (which, for convenience, we highlighted by underlining the value), the correlation results for the metrics are all statistically significant.

The pair-wise correlation between each source code metric and the mutation score is not strong. We speculate the reason behind the weak correlations to be collinearity of these code metrics. More specifically, Spearman's rank-order correlation analysis only evaluates the correlation between individual code metric and mutation score. Some code metrics could interact with each other. E.g., a long method is not necessary to have low mutation score. If there are more than four loops in a long method, then the method is very likely to have low mutation score.

**Table 7.** Spearman Results of Existing Code Metrics for Testability

| metric | rho | p-value | metric | rho | p-value | metric | rho | p-value |
|---|---|---|---|---|---|---|---|---|
| COMP | 0.0398 | 2.16E-08 | NOC | 0.1908 | 1.254E-161 | R-R(class) | **-0.2524** | 3.721E-285 |
| NOCL | 0.1047 | 2.32E-49 | NOA | 0.0423 | 2.723E-09 | NSUB(class) | -0.0048 | 0.5009 |
| NOS | -0.0139 | 0.05024 | CAST | -0.0162 | 0.02302 | NSUP(class) | **-0.2634** | 0 |
| HLTH | 0.0518 | 2.927E-13 | HDIF | 0.1334 | 2.691E-79 | NCO(class) | -0.0751 | 3.602E-26 |
| HVOC | 0.0485 | 8.831E-12 | NEXP | 0.0288 | 5.135E-05 | FOUT(class) | -0.1073 | 9.482E-52 |
| HEFF | 0.0856 | 1.595E-33 | NOMT(class) | 0.0981 | 1.564E-43 | DIT(class) | **-0.2634** | 0 |
| HBUG | 0.0518 | 3.163E-13 | LCOM(class) | 0.0564 | 2.125E-15 | CCOM(class) | 0.1695 | 1.589E-127 |
| CREF | 0.0193 | 0.00653 | AVCC(class) | 0.0405 | 1.206E-08 | COH(class) | 0.0001 | 0.9852 |
| XMET | 0.0465 | 5.743E-11 | NOS(class) | 0.0793 | 5.416E-29 | S-R(class) | 0.0016 | 0.8184 |
| LMET | -0.0221 | 0.00191 | HBUG(class) | 0.0824 | 3.826E-31 | MINC(class) | -0.0255 | 0.0003272 |
| NLOC | -0.0004 | 0.95 | HEFF(class) | 0.0982 | 1.213E-43 | EXT(class) | -0.0636 | 3.314E-19 |
| VDEC | 0.0281 | 7.702E-05 | UWCS(class) | 0.0929 | 3.708E-39 | INTR(class) | -0.0571 | 9.413E-16 |
| TDN | 0.0408 | 9.634E-09 | INST(class) | 0.0045 | 0.5238 | MPC(class) | -0.0636 | 3.314E-19 |
| NAND | 0.0357 | 5.191E-07 | PACK(class) | -0.1029 | 9.956E-48 | HVOL(class) | 0.0823 | 4.344E-31 |
| LOOP | 0.0685 | 5.116E-22 | RFC(class) | 0.095 | 6.38E-41 | HIER(class) | **-0.212** | 6.066E-200 |
| MOD | 0.0103 | 0.1482 | CBO(class) | -0.0157 | 0.0274 | HLTH(class) | 0.0911 | 9.53E-38 |
| NOPR | 0.067 | 3.801E-21 | MI(class) | 0.0482 | 1.144E-11 | SIX(class) | -0.197 | 2.388E-172 |
| EXCT | 0.1125 | 9.723E-57 | CCML(class) | 0.1559 | 6.998E-108 | TCC(class) | 0.0897 | 1.203E-36 |
| MDN | 0.053 | 8.3E-14 | NLOC(class) | 0.0756 | 1.692E-26 | NQU(class) | 0.1489 | 1.568E-98 |
| EXCR | -0.0067 | 0.3473 | RVF(class) | -0.033 | 3.498E-06 | F-IN(class) | 0.0875 | 6.031E-35 |
| HVOL | 0.0512 | 5.719E-13 | LCOM2(class) | -0.0486 | 7.691E-12 | MOD(class) | 0.0516 | 3.738E-13 |
| VREF | 0.0446 | 3.42E-10 | MAXCC(class) | -0.0178 | 0.01245 | LMC(class) | 0.1034 | 3.68E-48 |

**Table 8.** Spearman results of mutant observability metrics

| metric | rho | pvlaue | metric | rho | pvlaue |
|---|---|---|---|---|---|
| is_public | -0.0639 | 2.35E-19 | (cond(cond)) | -0.0415 | 5.4E-09 |
| is_static | 0.1137 | 6.29E-58 | (cond(loop)) | 0.0073 | 0.302 |
| is_void | -0.1427 | 1.42E-90 | (loop) | 0.0685 | 5.12E-22 |
| is_nested | 0.0466 | 5.38E-11 | (loop(cond)) | 0.0216 | 0.00242 |
| method_length | -0.0004 | 0.95 | (loop(loop)) | 0.0428 | 1.65E-09 |
| nested_depth | 0.053 | 8.3E-14 | non_void_percent | 0.2424 | 1.24E-262 |
| direct_test_no | **0.4177** | 0 | getter_percent | -0.153 | 6.23E-104 |
| test_distance | **-0.4921** | 0 | assertion-McCabe | **0.3956** | 0 |
| assertion_no | **0.3858** | 0 | assertion-density | **0.4096** | 0 |
| (cond) | 0.023 | 0.00124 | | | |

From Table 7, we can see that the highest *rho* is -0.2634 for both `NSUP(class)` standing for Number of Superclasses, and `DIT(class)`, or Depth of Inheritance Tree. Followed by `R-R(class)`, for Reuse Ratio, and `HIER(class)`, for Hierarchy method calls. At first glance, the top 4 metrics are all class-level metrics. However, we cannot infer that class-level metrics are more impactful on the mutation score than method-level ones. In particular, it can be related to the fact that we have considered more class-level metrics than method-level ones in the experiment. It would be an interesting direction for further researchers to investigate.

Besides, we expected that the metrics related to McCabe's Cyclomatic Complexity, i.e. `COMP`, `TCC`, `AVCC` and `MAXCC`, would show stronger correlation to the mutation score, as McCabe's Cyclomatic Complexity has been widely considered a powerful measure to quantify the complexity of a software program and it is used to provide a lower bound to the number of tests that should be written [47, 48, 49]). Based on our results without further investigation, we could only speculate that McCabe's Cyclomatic Complexity might not directly influence the mutation score. This could be another interesting angle to explore in future work.

**Summary** We found that the relation between the 64 existing source code quality metrics and the mutation score to be not so strong (<0.27).

### *5.1.2 Observability*

**Findings** Table 8 shows the overall results of Spearman's rank-order correlation analysis for mutant observability metrics. From Table 8, we can see that except for `method_length` and `(cond(loop))`, whose *p-value* is greater than 0.05, the results of the other observability metrics are statistically significant. The overall correlation between mutant observability metrics and mutation score are still not strong (<0.5), but significantly better than existing code metrics (<0.27). The top 5 are `test_distance`, `direct_test_no.`, `assertion-density`, `assertion-McCabe` and `assertion_no`. The metrics related to

**Table 9.** Random Forest Results of mutant observability metrics vs. Existing Metrics

| pid | ZeroR | | | | existing | | | | mutant observability | | | | combined | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | prec. | recall | AUC | err. | prec. | recall | AUC | err. | prec. | recall | AUC | err. | prec. | recall | AUC | err. |
| 1 | 0.856 | 0.856 | 0.497 | 0.2465 | 0.927 | 0.93 | 0.961 | 0.1014 | 0.940 | 0.942 | 0.960 | 0.0786 | **0.946** | **0.948** | **0.976** | **0.0741** |
| 2 | 0.913 | 0.913 | 0.498 | 0.1595 | 0.947 | 0.951 | 0.932 | 0.0775 | **0.960** | **0.962** | 0.946 | **0.063** | 0.957 | 0.959 | **0.951** | 0.067 |
| 3 | 0.815 | 0.815 | 0.499 | 0.3015 | 0.848 | 0.861 | 0.836 | 0.2039 | 0.866 | 0.864 | 0.871 | 0.1727 | **0.887** | **0.893** | **0.909** | **0.167** |
| 4 | 0.507 | 0.507 | 0.468 | 0.5001 | 0.667 | 0.667 | 0.733 | 0.3831 | **0.861** | **0.860** | **0.909** | **0.2044** | 0.827 | 0.827 | 0.887 | 0.2626 |
| 5 | 0.62 | 0.62 | 0.5 | 0.4712 | 0.842 | 0.843 | 0.908 | 0.2347 | 0.868 | 0.869 | 0.931 | 0.1801 | **0.901** | **0.901** | **0.955** | **0.168** |
| 6 | 0.726 | 0.726 | 0.493 | 0.3982 | 0.73 | 0.743 | 0.804 | 0.2948 | 0.708 | 0.716 | 0.779 | 0.2976 | **0.742** | **0.755** | **0.802** | **0.2946** |
| all | 0.569 | 0.569 | 0.5 | 0.4905 | 0.862 | 0.862 | 0.928 | 0.2133 | 0.864 | 0.864 | 0.937 | 0.1846 | **0.905** | **0.905** | **0.963** | **0.1625** |
| dir. | 0.853 | 0.853 | 0.499 | 0.2513 | 0.945 | 0.946 | 0.949 | 0.0915 | 0.941 | 0.943 | 0.955 | 0.0933 | **0.950** | **0.951** | **0.962** | **0.0886** |
| non. | 0.593 | 0.593 | 0.5 | 0.4829 | 0.853 | 0.853 | 0.923 | 0.2329 | 0.813 | 0.814 | 0.893 | 0.2371 | **0.878** | **0.879** | **0.941** | **0.2075** |

**Table 10.** Feature Importances of Classification Model (1)

| 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|
| metric | imp. | metric | imp. | metric | imp. | metric | imp. | metric | imp. |
| test_distance | 0.35 | test_distance | 0.15 | test_distance | 0.13 | test_distance | 0.48 | test_distance | 0.23 |
| NLOC(class) | 0.15 | HIER(class) | 0.12 | NOCL | 0.05 | method_length | 0.03 | is_void | 0.1 |
| NOCL | 0.03 | CCML(class) | 0.05 | HDIF | 0.03 | COMP | 0.03 | EXCT | 0.04 |
| CREF | 0.03 | NLOC(class) | 0.05 | MI(class) | 0.03 | NOCL | 0.03 | NOCL | 0.03 |
| MINC(class) | 0.03 | NOCL | 0.04 | is_static | 0.02 | CAST | 0.03 | NOS | 0.03 |
| non_void_percent | 0.02 | MI(class) | 0.04 | non_void_percent | 0.02 | HDIF | 0.03 | S-R(class) | 0.03 |
| HDIF | 0.02 | assertion-density | 0.03 | HVOC | 0.02 | (cond) | 0.02 | is_public | 0.02 |
| NOS(class) | 0.02 | CREF | 0.03 | HEFF | 0.02 | VREF | 0.02 | nested_depth | 0.02 |
| PACK(class) | 0.02 | HDIF | 0.03 | CREF | 0.02 | is_void | 0.01 | direct_test_no | 0.02 |
| TCC(class) | 0.02 | PACK(class) | 0.03 | VREF | 0.02 | direct_test_no | 0.01 | assertion_no | 0.02 |
| LMC(class) | 0.02 | method_length | 0.02 | NEXP | 0.02 | assertion_no | 0.01 | CREF | 0.02 |
| HLTH | 0.01 | HVOC | 0.02 | HEFF(class) | 0.02 | non_void_percent | 0.01 | HDIF | 0.02 |
| HVOC | 0.01 | HEFF | 0.02 | PACK(class) | 0.02 | assertion-density | 0.01 | PACK(class) | 0.02 |
| HEFF | 0.01 | LMET | 0.02 | CBO(class) | 0.02 | HLTH | 0.01 | F-IN(class) | 0.02 |
| XMET | 0.01 | NOA | 0.02 | CCML(class) | 0.02 | HVOC | 0.01 | method_length | 0.01 |

*test directness*, i.e., `test_distance` (-0.4923) and `direct_test_no` (0.4177) are ranked first in terms of *rho* among all metrics that we consider (including existing code metrics in Section 2.2). This observation corresponds to our findings in Section 3 and our expectations that the methods with no direct test are more difficult for mutation testing. In terms of *rho* values, the assertion related metrics are ranked after test directness related metrics, which supports our conjectures in Section 3 that the quality of assertions can influence the outcome of mutation testing.

**Summary** The correlations between mutant observability metrics and mutation score are not very strong (<0.5), however, significantly better than the correlations for existing code metrics. Test directness (`test_distance` and `direct_test_no.`) takes the first place of `NSUP(class)` in *rho* among all metrics (including existing ones in Section 2.2), followed by assertion-based metrics (`assertion-density`, `assertion-McCabe` and `assertion_no`).

## 5.2 Random forests

**Classification effectiveness** As discussed in Section 4.3, we compare the four models in terms of both our mutant observability metrics and the existing metrics, i.e.,

1. `ZeroR`: model using ZeroR approach
2. `existing`: random forests model based on existing code metrics
3. `mutant observability`: random forests model based on mutant observability metrics
4. `combined`: random forests model based on the combination of existing metrics and mutant observability metrics.

The comparison of the four models are shown in Table 9. To make clear which model performs better than the others, we highlighted the values of the model achieving the best performance among the four in **bold**, that of second best in underline. For precision, recall and AUC, the model with best performance

**Table 11.** Feature Importances of Classification Model (2)

| 6 | | all | | dir. | | non-dir. | |
|---|---|---|---|---|---|---|---|
| metric | imp. | metric | imp. | metric | imp. | metric | imp. |
| CBO(class) | 0.09 | test_distance | 0.29 | is_void | 0.22 | test_distance | 0.16 |
| HDIF | 0.07 | PACK(class) | 0.06 | PACK(class) | 0.13 | NOCL | 0.09 |
| NQU(class) | 0.06 | NOCL | 0.05 | HDIF | 0.05 | non_void_percent | 0.04 |
| test_distance | 0.04 | is_void | 0.03 | NOS | 0.04 | EXCT | 0.04 |
| non_void_percent | 0.03 | EXCT | 0.03 | assertion-density | 0.03 | HDIF | 0.03 |
| HVOC | 0.03 | non_void_percent | 0.02 | NEXP | 0.03 | PACK(class) | 0.03 |
| HEFF | 0.03 | CREF | 0.02 | direct_test_no | 0.02 | MI(class) | 0.03 |
| CREF | 0.03 | HDIF | 0.02 | assertion_no | 0.02 | CREF | 0.02 |
| XMET | 0.03 | MI(class) | 0.02 | assertion-McCabe | 0.02 | CBO(class) | 0.02 |
| NAND | 0.03 | is_public | 0.01 | NOCL | 0.02 | MINC(class) | 0.02 |
| VREF | 0.03 | is_nested | 0.01 | CREF | 0.02 | HIER(class) | 0.02 |
| NOA | 0.03 | method_length | 0.01 | NOA | 0.02 | F-IN(class) | 0.02 |
| NEXP | 0.03 | nested_depth | 0.01 | MINC(class) | 0.02 | MOD(class) | 0.02 |
| method_length | 0.02 | assertion_no | 0.01 | method_length | 0.01 | is_public | 0.01 |
| NOCL | 0.02 | getter_percent | 0.01 | nested_depth | 0.01 | is_static | 0.01 |

is the one with the highest value, while for the mean absolute error, the best scoring model exhibits the lowest value.

From Table 9, we can see that the random forest models are better than the baseline `ZeroR` which only relies on the majority. This is the *prerequisite* for further comparison. `Combined` achieves the best performance (in 5 out of 6 projects) compared to the existing code metrics and mutant observability metrics in terms of AUC; this observation is as expected since `combined` considered both the existing and our metrics during training, which provides the classification model with more information. The only exception is `java-apns-apns-0.2.3`. We conjecture that the number of instances (selected methods) in this project might be too small (only 150) to develop a sound prediction model. In second place comes the model based on mutant observability metrics, edging out the model based on existing metrics.

For the overall dataset (the 7th row marked with "all" in Table 9), `combined` takes the first place in all evaluation metrics. In second place come the `mutant observability`, slightly better than `existing`. Another angle which is interesting to further investigate is the *test directness*. If we only consider the methods that are directly tested (the second to last row in Table 9), `combined` again comes in first, followed by the existing code metrics model. The same observation holds for the methods that are not directly tested (the last row in Table 9). It is easy to understand that when the dataset only considers methods that are directly tested (or not), the test directness features in our model become irrelevant. However, we can see that the difference between existing metrics and ours are quite tiny (<3.4%).

**Feature importances analysis** Tables 10 and 11 show the top 15 features per project (and overall) in descending order. We can see that for five out of the the six projects (including the overall dataset), `test_distance` ranks first. This again supports our previous findings that *test directness* plays a significant role in mutation testing. The remaining features in the top 14 vary per projects; this is not surprising, as the task and context of these projects are varying greatly. For example, `Apache Commons Lang` (Column "2" in Table 10) is a utility library that provides a host of helper methods for the `java.lang API`. Therefore, most methods in `Apache Commons Lang` are public and static; thus `is_public` and `is_static` are not among the top 15 features for `Apache Commons Lang`. A totally different context is provided by the `JFreeChart` project (Column "5" in Table 10). `JFreeChart` is a Java chart library, whose class encapsulation and inheritance hierarchy are well-designed, so `is_public` appears among the top 15 features.

Zooming in on the overall dataset (Column "all" in Table 11), there are eight metrics from our proposed mutant observability metrics among the top 15 features. The importance of `test_distance` is much higher than the other features (¿4.83X). In second place comes `PACK(class)`, or the number of packages imported. This observation is easy to understand since `PACK(class)` denotes the complexity of dependency, and dependency could influence the difficulty of testing, especially when making use of mocking objects. Thereby, dependency affects the mutation score. Clearly, more investigations are required to draw further conclusions. The third place in the feature importance analysis is taken by `NOCL`, which stands for the Number of Comments. This observation is quite interesting since `NOCL` is related to how hard it is to understand the code (*code readability*). This implies that code readability could have an impact on mutation testing. It is certainly an invitation for future work to explore the relationship between

**Table 12.** Selected feature by PCA

| | | | |
|---|---|---|---|
| is_public | (cond) | assertion-density | XMET |
| is_static | (cond(cond)) | COMP | LMET |
| is_void | (cond(loop)) | NOCL | NLOC |
| is_nested | (loop) | NOS | VDEC |
| method_length | (loop(cond)) | HLTH | TDN |
| nested_depth | (loop(loop)) | HVOC | NAND |
| direct_test_no | non-void_percent | HEFF | LOOP |
| test_distance | getter_percent | HBUG | MOD |
| assertion_no | assertion-McCabe | CREF | NOPR |

code readability and mutation testing.

As for the methods with direct tests (Column "dir." in Table 11), `is_void` takes the first position, which indicates that it is more difficult to achieve a high mutation score for void methods. Considering the methods without direct tests (Column "non-dir." in Table 11), `test_distance` again ranks first.

Another interesting observation stems from the comparison of the performance of assertion related metrics in the feature importance analysis and the Spearman rank order correlation results (in Section 5.1). For Spearman's rank order correlation, we can see that assertion related metrics are the second significant category right after test directness (in Table 8 in Section 5.1). While in the feature importance analysis, assertion related metrics mostly rank after the top 5 (shown in Table 10 and Table 11) We speculate that the major reason is because test directness and assertion related metrics are almost *collinear* in the prediction model. For the six subjects, there are no tests without assertions. If the method has a direct test, then the corresponding assertion no. is always greater than 1. Therefore, the ranks of assertion related metrics are not as high as we had initially expected in the feature importance analysis.

**Summary** Overall, the random forests model based on the combination of existing code metrics and mutant observability metrics performs best, followed by that on mutant observability metrics. The analysis of feature importances shows that *test directness* ranks highest, remarkably higher than the other metrics.

## 6 RQ4 CODE REFACTORING

It is our goal to investigate whether we can refactor away the observability issue that we expect to hinder tests from killing mutants and thus to affect the mutation score. In an in-depth case study, we manually analyse 16 code fragments to better understand the interaction between observability, the metrics that we have been investigating, and the possibilities for refactoring.

Our analysis starts from the `combined` model, which as Table 9 shows, takes the leading position among the models. We then apply *Principal Component Analysis (PCA)* [50] to perform feature selection, which, as Table 12 shows, leaves us with 36 features (or metrics). Then, as discussed in Section 4, we build a decision tree based on those 36 metrics using J48 (shown in Figure 3), and select the top 6 leaves (also called end nodes) in the decision tree for further *manual* analysis as potential refactoring guidelines.

Here, we take a partial decision tree to demonstrate how we generate rules (shown in Figure 4). In Figure 4, we can see that there are three attributes (marked as ellipse) and four end nodes (marked as rectangle) in the decision tree. Since we would like the investigate how code refactoring improves mutation score (**RQ4**), we only consider the end nodes labeled with "LOW" denoting mutation score<0.5. By combining the conditions along the paths of the decision tree, we obtain the two rules for "LOW" end nodes (as shown in the first column of the table in Figure 4). For every end node, there are two values attached to the class: the first is the number of instances that correctly fall into the node, the other is the instances that incorrectly fall into the node. The accuracy in the table is computed by the number of correct instances divided by that of total instances. As mentioned earlier, we select the top 6 end nodes from the decision tree, where the end nodes are ranked by the number of correct instances under the condition accuracy$\geq$0.8.

In our *actual* case study, we manually analyse 16 cases in total. Due to space limitations, we only highlight six cases in this paper (all details are available on GitHub [36]). We will discuss our findings in code refactoring case by case.

### 6.1 Case 1: `org.jfree.chart.plot.MeterPlot::drawValueLabel`

This case (shown in Listing 4) is under anti-pattern **Rule 1**: `test_distance > 5 && (loop(loop))` $\leq$ `0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop)` $\leq$ `0 && NOCL` $\leq$ `9 &&`

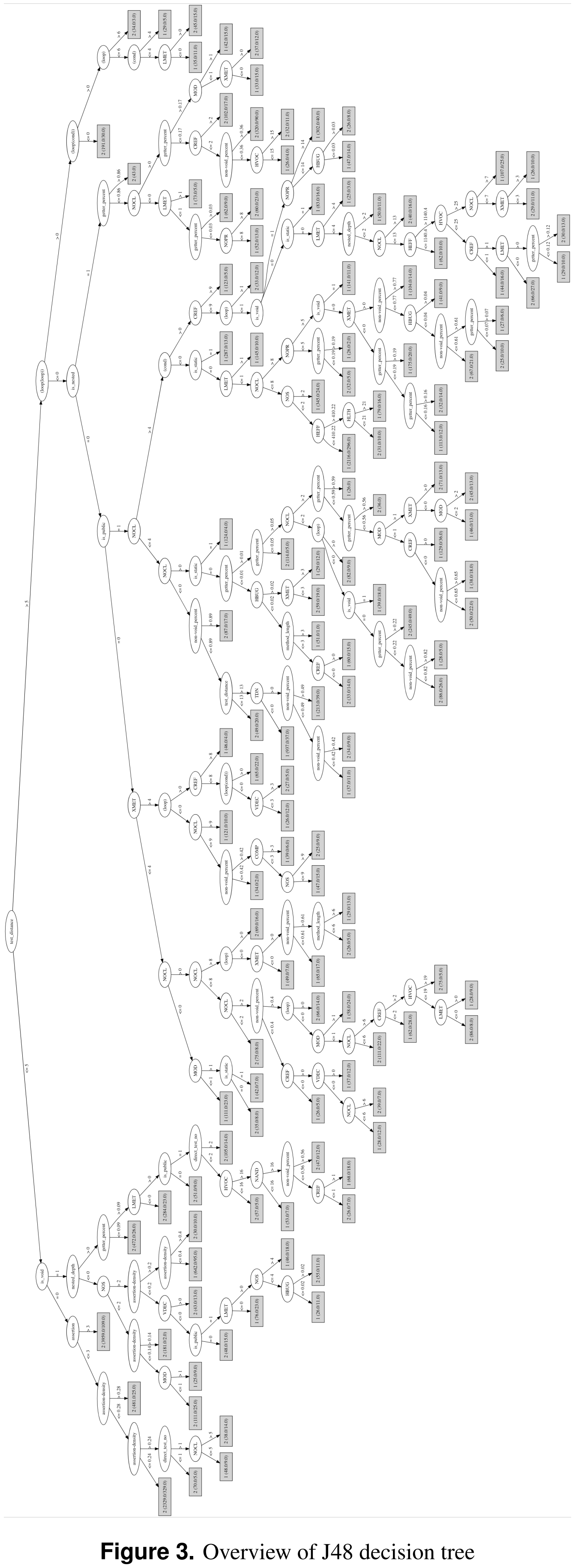
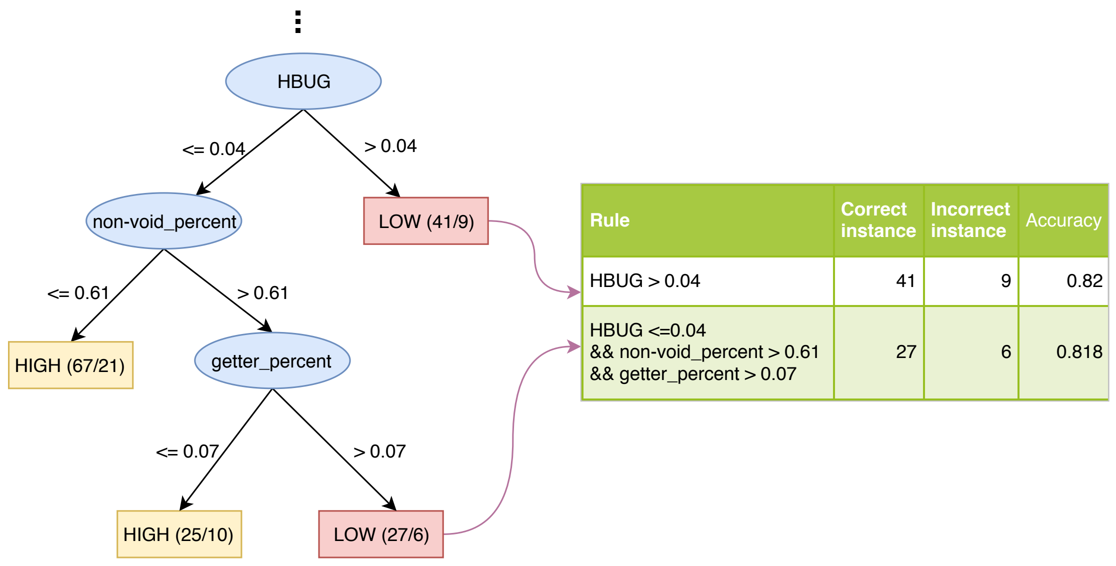


**Figure 3.** Overview of J48 decision tree



| Rule | Correct instance | Incorrect instance | Accuracy |
|---|---|---|---|
| HBUG > 0.04 | 41 | 9 | 0.82 |
| HBUG <=0.04 && non-void_percent > 0.61 && getter_percent > 0.07 | 27 | 6 | 0.818 |

**Figure 4.** Demo of rule generation

**Table 13.** Summary of mutants from Listing 4 (Case 1)

| ID | Line No. | Mutator | Results |
|---|---|---|---|
| 1 | 1146 | removed call to java/awt/Graphics2D::setFont | SURVIVED |
| 2 | 1147 | removed call to java/awt/Graphics2D::setPaint | SURVIVED |
| 3 | 1149 | negated conditional | SURVIVED |
| 4 | 1151 | negated conditional | SURVIVED |
| 5 | 1157 | Replaced float addition with subtraction | SURVIVED |

**Table 14.** Summary of mutants from Listing 8 (Case 2)

| ID | Line No. | Mutator | Results |
|---|---|---|---|
| 1 | 165 | mutated return of Object value for org/jfree/chart/util/PaintAlpha::darker to ( if (x != null) null else throw new RuntimeException ) | NO_COVERAGE |
| 2 | 166 | Replaced double multiplication with division | NO_COVERAGE |
| 3 | 167 | Replaced double multiplication with division | NO_COVERAGE |
| 4 | 168 | Replaced double multiplication with division | NO_COVERAGE |

`non-void_percent` $\leq$ `0.42`. In total, there are 5 mutants generated from this method (shown in Table 13). All 5 mutants survive the test suite.

**Code refactoring** We start with `test_distance > 5` which means there is no direct test for this method. Accordingly, we add one direct test (shown in Listing 5).

However, Mutant 4 and 5 cannot be killed by adding the above direct test. Upon inspection, we found that Mutant 4 and 5 cannot be killed because the `DrawValueLabel(...)` method is void. In particular, this means that the changes in state caused by the `TextUtils.drawAlignedString()` method (line 1158) cannot be assessed. This is indicated by `non-void_percent` $\leq$ `0.42` in **Rule 1**. We then refactor the method to have it return `Rectangle2D` (shown in Listing 6). Also, we improve the direct test for this method in Listing 5 by adding a new test method (shown in Listing 7) in order to avoid the *assertion roulette* test smell [51, 52]. By refactoring the method to non-void and adding a direct test, all previously surviving mutants are now successfully killed.

### 6.2 Case 2: `org.jfree.chart.axis.SymbolAxis::drawGridBands`

This case (shown in Listing 8) is under **Rule 2**: `test_distance > 5 && (loop(loop))` $\leq$ `0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop)` $\leq$ `0 && NOCL > 9`. In total, 4 mutants are generated from this method (see Table 14). None of the mutants are killed.

```
/**
 * Draws the value label just below the center of the dial.
 *
 * @param g2  the graphics device.
 * @param area  the plot area.
 */
protected void drawValueLabel(Graphics2D g2, Rectangle2D area) {
    g2.setFont(this.valueFont);
    g2.setPaint(this.valuePaint);
    String valueStr = "No value";
    if (this.dataset != null) {
        Number n = this.dataset.getValue();
        if (n != null) {
            valueStr = this.tickLabelFormat.format(n.doubleValue()) + " "
                     + this.units;
        }
    }
    float x = (float) area.getCenterX();
    float y = (float) area.getCenterY() + DEFAULT_CIRCLE_SIZE;
    TextUtils.drawAlignedString(valueStr, g2, x, y,TextAnchor.TOP_CENTER);
}
```

**Listing 4.** `org.jfree.chart.plot.MeterPlot::drawValueLabel` (Case 1)

```
@Test
 public void testDrawValueLabel(){
     MeterPlot p1 = new MeterPlot(new DefaultValueDataset(1.23));
     BufferedImage image = new BufferedImage(3, 4, BufferedImage.TYPE_INT_ARGB);
     Graphics2D g2 = image.createGraphics();
     Rectangle2D area = new Rectangle(0, 0, 1, 1);
     p1.drawValueLabel(g2,area);
     assertTrue(g2.getFont() == p1.getValueFont());
     assertTrue(g2.getPaint() == p1.getValuePaint());
 }
```

**Listing 5.** Direct test for Listing 4 (Case 1)

**Code refactoring** It is clear that this method is private, thus it is impossible to directly call this method from outside the class. We first refactor this method from `private` to `public`. This is revealed by `is_public = 0` in **Rule 2**.

Then, guided by `test_distance > 5` from **Rule 2**, we add a direct test for this method to kill all mutants (see Listing 10).

### 6.3 Case 3: `org.apache.commons.lang3.builder.IDKey::hashCode`

This case (shown in Listing 11) is under **Rule 3**: `test_distance > 5 && (loop(loop))` $\leq$ `0 && is_nested = 0 && is_public = 1 && NOCL` $\leq$ `4 && NOCL > 0 && is_static = 0 && getter_percent` $\leq$ `0.01 && HBUG` $\leq$ `0.02 && method_length > 3`. Only one mutant is generated for this method: a mutant that replaces the return value with (x == 0 ? 1 : 0). This mutant survives.

**Code refactoring** Starting with `test_distance > 5`, we add a direct test for this method (shown in Listing 12), which works perfectly to kill the mutant.

### 6.4 Case 4: `org.jfree.chart.renderer.category.AbstractCategoryItemRenderer::drawOutline`

This case (shown in Listing 13) is under **Rule 4**: `test_distance > 5 && (loop(loop))` $\leq$ `0 && is_nested = 0 && is_public = 1 && NOCL > 4 && (cond)` $\leq$ `0 && is_static = 0 && LMET` $\leq$ `1 && NOCL > 8 && NOPR > 5 && is_void = 1`. Also in this case, only 1 mutant is generated for this method. The particular change applied is the removal of the call to `org.jfreechart.plot.CategoryPlot::drawOutline`. The mutant is not killed by the original test suite.

```
protected Rectangle2D drawValueLabel(Graphics2D g2, Rectangle2D area) {
    g2.setFont(this.valueFont);
    g2.setPaint(this.valuePaint);
    String valueStr = "No value";
    if (this.dataset != null) {
        Number n = this.dataset.getValue();
        if (n != null) {
            valueStr = this.tickLabelFormat.format(n.doubleValue()) + " "
                     + this.units;
        }
    }
    float x = (float) area.getCenterX();
    float y = (float) area.getCenterY() + DEFAULT_CIRCLE_SIZE;
    return TextUtils.drawAlignedString(valueStr, g2, x, y,TextAnchor.TOP_CENTER);
}
```

**Listing 6.** Refactoring of Listing 4 (Case 1)

```
@Test
public void testDrawValueLabelArea() {
    MeterPlot p1 = new MeterPlot(new DefaultValueDataset(1.23));
    BufferedImage image = new BufferedImage(3, 4, BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2 = image.createGraphics();
    Rectangle2D area = new Rectangle(0, 0, 1, 1);
    Rectangle2D drawArea = p1.drawValueLabel(g2,area);
    assertEquals(0.5,drawArea.getCenterX(),0.01);
    assertEquals(18.8671875,drawArea.getCenterY(),0.01);
    assertEquals(15.0,drawArea.getHeight(),0.01);
    assertEquals(64.0,drawArea.getWidth(),0.01);
}
```

**Listing 7.** Improved direct test for Listing 4 (Case 1)

**Code refactoring** Based on `test_distance > 5`, we add one direct test (as shown in Listing 14) for this method to kill the surviving mutant.

### 6.5 Case 5: `org.jfree.chart.renderer.category.AbstractCategoryItemRenderer::drawOutline`

This case (shown in Listing 15) is under **Rule 5**: `test_distance` $\leq$ `5 && is_void = 1 && nested_depth` $\leq$ `0 && NOS` $\leq$ `2 && assertion-density` $\leq$ `0.14 && MOD > 1`. In this case a single (surviving) mutant is generated that removes the call to `org.apache.commons/lang3.builder.ToStringStyle::setUseShortClassName`.

**Code refactoring** We can see that **Rule 5** is different from the previous rule in that `test_distance` is less than 5, while in **Rule 4** `test_distance > 5`. A more in-depth analysis reveals that the method in Listing 15 is already directly invoked by the original test suite. The surviving mutant is due to the fact that there are no assertions that examine the changes after the `setUseShortClassName` method call. This situation is reflected by `assertion-density` $\leq$ `0.14` in **Rule 5**. Therefore, we add assertions to assess the changes (seen in Listing 16), which leads to the mutant being killed.

### 6.6 Case 6: `org.apache.commons.math3.exception.TooManyEvaluationsException::<init>`

This case (shown in Listing 17) is under **Rule 6**: `test_distance` $\leq$ `5 && is_void = 1 && nested_depth` $\leq$ `0 && NOS > 2 && assertion-density` $\leq$ `0.22 && CREF > 1 && XMET > 0 && VDEC` $\leq$ `0 && NOCL` $\leq$ `12`. A single mutant is generated: a removal of the call to `org.apache.commons.math3.exception.util.ExceptionContext::addMessage`. This mutant is surviving the test suite.

**Code refactoring** We found that the mutant in Line 37 cannot be killed because the function `addMessage` changes the field `List<Localizable> msgPatterns`. This field is private in the class `ExceptionContext`

```
/**
 * Similar to {@link Color#darker()}.
 * <p>
 * The essential difference is that this method
 * maintains the alpha-channel unchanged<br>
 *
 * @param paint a {@code Color}
 *
 * @return a darker version of the {@code Color}
 */
private static Color darker(Color paint) {
    return new Color(
            (int)(paint.getRed  () * FACTOR),
            (int)(paint.getGreen() * FACTOR),
            (int)(paint.getBlue () * FACTOR), paint.getAlpha());
}
```

**Listing 8.** `org.jfree.chart.axis.SymbolAxis::drawGridBands` (Case 2)

```
public static Color darker(Color paint) {
    return new Color(
            (int)(paint.getRed  () * FACTOR),
            (int)(paint.getGreen() * FACTOR),
            (int)(paint.getBlue () * FACTOR), paint.getAlpha());
}
```

**Listing 9.** Refactoring of Listing 8 (Case 2)

and there is no other way to access it. As such, our first step is to add a getter for `msgPatterns` (shown in Listing 18). In **Rule 6**, we can see that `is_void = 1` is the underlying cause since void methods could be difficult to test if no getters for private fields exist.

To kill the surviving mutant, we add one extra assertion (in a new test method) to examine the changes in `msgPatterns` (in Listing 19). This action is also partly evidenced by `assertion-density` $\leq$ `0.22` in **Rule 6**. As assertion-density denotes the ratio between the total number of assertions in direct tests and the lines of code in direct tests, low assertion-density is a sign of insufficient assertions in the direct tests to detect the mutant.

## 6.7 RQ4 Summary

Based on all 16 cases that we analysed (available in our GitHub repository [36]), we found that our mutant observability metrics can lead to actionable refactorings that enable to kill mutants that were previously not being killed. Ultimately, this leads to an improvement of the mutation score:

- most cases can be easily fixed by adding direct tests if `test_distance>5`.
- most cases can be easily fixed by adding assertions if `test_distance`$\leq$`5`.
- private methods must be refactored to protected/public for testing (indicated by `is_public=0`).

```
@Test
public void testDarker(){
    Color paint = new Color(10,20,30);
    Color darker = PaintAlpha.darker(paint);
    assertEquals(7,darker.getRed());
    assertEquals(14,darker.getGreen());
    assertEquals(21,darker.getBlue());
}
```

**Listing 10.** Direct test for Listing 8 (Case 2)

```
/**
 * returns hash code - i.e. the system identity hashcode.
 * @return the hashcode
 */
@Override
public int hashCode() {
   return id;
}
```

**Listing 11.** `org.apache.commons.lang3.builder.IDKey::hashCode` (Case 3)

```
@Test
public void testHashCode(){
    IDKey idKey = new IDKey(new Integer(123));
    assertEquals(989794870,idKey.hashCode());
}
```

**Listing 12.** Direct test for Listing 11 (Case 3)

- three void methods had to be refactored to be non-void (indicated by `is_void = 1` and `non-void_percent` $\leq$ `0.42`).
- one void method needed an additional getter because a private field was changed (indicated by `is_void = 1`).

## 7 THREATS TO VALIDITY

**External validity** Our results are based on mutants generated by the operators implemented in PiTest. While PiTest is a frequently used mutation testing tool, our results might be different when using other mutation tools [53]. With regard to the subject systems selection, we chose six open-source projects from GitHub; the selected projects differ in size, number of test cases and application domain. Nevertheless, we do acknowledge that a broad replication of our study would mitigate any generalizability concerns even further.

**Internal validity** The main threat to internal validity for our study is the implementation of the MUTATION OBSERVER tool for the experiment. To reduce internal threats to a large extent, we relied on existing tools that have been widely used, e.g., WEKA, MATLAB and PiTest. Moreover, we carefully reviewed and tested all code for our study to eliminate potential faults in our implementation. Another threat to internal validity is the disregard of equivalent mutants in our experiment. However, this threat is unavoidable and shared by other studies on mutation testing that attempt to detect equivalent mutants or not. Moreover, we consider equivalent mutants as potential weakness in the software (reported by Coles [54, slide 44-52]),

```
/**
 * Draws an outline for the data area.  The default implementation just
 * gets the plot to draw the outline, but some renderers will override this
 * behaviour.
 *
 * @param g2  the graphics device.
 * @param plot  the plot.
 * @param dataArea  the data area.
 */
@Override
public void drawOutline(Graphics2D g2, CategoryPlot plot,
        Rectangle2D dataArea) {
    plot.drawOutline(g2, dataArea);
}
```

**Listing 13.**
`org.jfree.chart.renderer.category.AbstractCategoryItemRenderer::drawOutline` (Case 4)

```
@Test
public void testDrawOutline(){
    AbstractCategoryItemRenderer r = new LineAndShapeRenderer();
    BufferedImage image = new BufferedImage(200 , 100,
            BufferedImage.TYPE_INT_RGB);
    Graphics2D g2 = image.createGraphics();
    CategoryPlot plot = new CategoryPlot();
    Rectangle2D dataArea = new Rectangle2D.Double();
    r.drawOutline(g2,plot,dataArea);
    assertTrue(g2.getStroke()==plot.getOutlineStroke());
}
```

**Listing 14.** Direct test for Listing 13 (Case 4)

```
/**
 * <p>Sets whether to output short or long class names.</p>
 *
 * @param useShortClassName  the new useShortClassName flag
 * @since 2.0
 */
@Override
public void setUseShortClassName(final boolean useShortClassName) { // NOPMD as
    this is implementing the abstract class
    super.setUseShortClassName(useShortClassName);
}
```

**Listing 15.**
`org.jfree.chart.renderer.category.AbstractCategoryItemRenderer::drawOutline` (Case 5)

thereby, we did not manually detect equivalent mutants in this paper.

**Construct validity** The main threat to construct validity is the measurement we used to evaluate our methods. We minimise this risk by adopting evaluation metrics that are widely used in research (such as recall, precision and AUC), as well as a sound statistical analysis to assess the significance (Spearman's rank-order correlation).

## 8 RELATED WORK

The notion of *software testability* dates back to 1991 when Freedman [28] formally defined *observability* and *controllability* in the domain of software. Voas [29] proposed a dynamic technique coined propagation, infection, and execution (PIE) analysis for statistically estimating the program's *fault sensitivity*. More recently, researchers have aimed to increase our collective understanding of *testability* by using statistical methods to predict *testability* based on various code metrics. An prime example is the work of Bruntink and van Deursen [8], who have explored the relation between nine class-level object-oriented metrics and testability. To the best of our knowledge, there is no study that uses statistical methods to investigate the relation between code quality metrics based on *testability* and *observability* and mutation score.

Mutation testing was initially introduced as a fault-based testing method which was regarded as significantly better at detecting errors than the *covering measure* approach [55]. Since then, mutation

```
@Test
public void testSetUseShortClassName(){
    assertTrue(STYLE.isUseShortClassName());
    STYLE.setUseShortClassName(false);
    assertFalse(STYLE.isUseShortClassName());
    STYLE.setUseShortClassName(true);
    assertTrue(STYLE.isUseShortClassName());
}
```

**Listing 16.** Additional assertions for Listing 15 (Case 5)

```
/**
 * Construct the exception.
 *
 * @param max Maximum number of evaluations.
 */
public TooManyEvaluationsException(Number max) {
    super(max);
    getContext().addMessage(LocalizedFormats.EVALUATIONS);
}
```

**Listing 17.** `org.apache.commons.math3.exception.TooManyEvaluationsException::<init>` (Case 6)

```
public List<Localizable> getMsgPatterns(){
    return msgPatterns;
}
```

**Listing 18.** Refactoring of Listing 17 (Case 6)

testing has been actively investigated and studied thereby resulting in remarkable advances in its concepts, theory, technology and empirical evidence. For more literature on mutation testing, we refer to the existing surveys of DeMillo [56], Offutt and Untch [57], Jia and Harman [1], Offutt [11] and Zhu et al. [12]. Here we mainly address the studies that concern *mutant utility* [22], the efficacy of mutation testing. Yao et al. [27] have reported on the causes and prevalence of equivalent mutants and their relationship to stubborn mutants based on a manual analysis of 1230 mutants. Just et al. [22] have shown a strong correlation between mutant utility and context information from the program in which the mutant is embedded. Brown et al. [24] have developed a method for creating potential faults that are more closely coupled with changes made by actual programmers where they named “wild-caught mutants”. Chekam et al. [58] have investigated the problem of selecting the fault revealing mutants. They put forward a machine learning approach (decision trees) that learns to select fault revealing mutants from a set of static program features. Jimenez et al. [26] investigated the use of natural language modelling techniques in mutation testing. All above studies have enriched the understanding of mutation testing, especially its efficacy. However, the aim of our work is different from those studies, as we would like to gain insights into how code quality affects the efforts needed for mutation testing.

The study most related to ours is that of Zhang et al.’s *predictive mutation testing*, where they have constructed a classification model to predict mutant killable result based on a series of features related to mutants and tests. In their discussion, they compared source code related features and test code related features in the prediction model for the mutation score. They found that test code features are more important than source code ones. But from their results, we cannot draw clear conclusions on the impact of code quality on mutation testing as their goal is to predict exact killable mutant results. Another interesting work close to our study is Vera-Pérez et al. [59]’s *pseudo-tested methods*. Pseudo-tested methods denote those methods that are covered by the test suite, but for which no test case fails even if the entire method body is completely stripped. They rely on the idea of “extreme mutation”, which completely strips out the body of a method. The difference between Vera-Pérez et al. [59]’s study and

```
@Test
public void testMsgPatterns() {
    final int max = 12345;
    final TooManyEvaluationsException e = new TooManyEvaluationsException(max);
    final String msg = e.getLocalizedMessage();
    Assert.assertTrue(e.getContext().getMsgPatterns()
                .contains(LocalizedFormats.EVALUATIONS));
}
```

**Listing 19.** Additional assertion for Listing 17 (Case 6)

ours is that we pay attention to *conventional* mutation operators rather than “extreme mutation”.

## 9 CONCLUSION & FUTURE WORK

This paper aims to increase our understanding of the mutation score, especially to investigate the relationship between *testability* and *observability* metrics and the mutation score. More specifically, we have collected 64 existing source code quality metrics for testability, and have proposed a set of metrics that specifically target *observability*. The results from our empirical study involving 6 open-source projects show that the 64 existing code quality metrics are not strongly correlated with the mutation score ($rho<0.27$). In contrast, the 19 newly proposed *mutant observability metrics*, that are defined in terms of both production code and test cases, do show stronger correlation with the mutation score ($rho<0.5$). In particular, *test directness*, `test_distance` and `direct_test_no` stand out.

In order to better understand the causality of our insights, we continued our investigation with a manual analysis of 16 methods that scored particularly bad in terms of mutation score, i.e., a number of mutants were not killed by the existing tests. In particular, we have refactored these methods according to the anti-patterns that we established in terms of the mutant observability metrics. Our aim here was to establish whether the removal of the observability anti-patterns would lead to an improvement of the mutation score. We found that these anti-patterns can indeed offer software engineers actionable insights to improve both the production code and the test suite, and improve the mutation score along with it. For instance, we found that private methods (expressed as `is_public=0` in our schema) are prime candidates to potentially refactor to increase their observability, e.g., by making them public or protected for testing purpose.

Our approach is implemented in a prototype tool (coined MUTATION OBSERVER, openly available on GitHub [36]) that automatically calculates mutant observability metrics. With our tool, and since the results are encouraging, we envision the following future work: 1) conduct additional empirical studies on more subject systems; 2) evaluate the usability of our mutant observability metrics by involving practitioners; 3) investigate the relations between more code metrics (e.g., code readability) and mutation score.

## ACKNOWLEDGMENTS

This work has been partially funded by the Netherlands Organisation for Scientific Research (NWO) through the “TestRoots” project. Further funding came from the EU Horizon 2020 ICT-10-2016-RIA “STAMP” project (No.731529).

## REFERENCES

[1] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[2] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just, “An industrial application of mutation testing: Lessons, challenges, and research directions,” in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICST Workshops)*, pp. 47–53, IEEE, 2018.

[3] H. Coles, “GitHub Repository for PIT.” `https://github.com/hcoles/pitest`. [Online; accessed 16-October-2018].

[4] H. Coles, “PIT Main Page.” `http://pitest.org/`. [Online; accessed 16-October-2018].

[5] G. Petrovic and M. Ivankovic, “State of mutation testing at Google,” in *Proceedings of the International Conference on Software Engineering in Practice (ICSE SEIP)*, 2018.

[6] Q. Zhu and A. Zaidman, “Mutation testing for physical computing,” in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 289–300, IEEE, 2018.

[7] I. ISO, “Iso 9126/iso, iec (hrsg.): International standard iso/iec 9126: Information technology-software product evaluation,” *Quality Characteristics and Guidelines for their use*, pp. 12–15, 1991.

[8] M. Bruntink and A. van Deursen, “An empirical study into class testability,” *Journal of systems and software*, vol. 79, no. 9, pp. 1219–1232, 2006.

[9] M. Staats, M. W. Whalen, and M. P. Heimdahl, “Better testing through oracle selection (nier track),” in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 892–895, ACM, 2011.

[10] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[11] J. Offutt, “A mutation carol: Past, present and future,” *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[12] Q. Zhu, A. Panichella, and A. Zaidman, “A systematic literature review of how mutation testing supports quality assurance processes,” *Software Testing, Verification and Reliability*, vol. 28, no. 6, p. e1675, 2018. e1675 stvr.1675.

[13] A. P. Mathur and W. E. Wong, “An empirical comparison of data flow and mutation-based test adequacy criteria,” *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[14] P. G. Frankl, S. N. Weiss, and C. Hu, “All-uses vs mutation testing: an experimental comparison of effectiveness,” *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[15] N. Li, U. Praphamontripong, and J. Offutt, “An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage,” in *Software Testing, Verification and Validation Workshops, 2009. ICSTW’09. International Conference on*, pp. 220–229, IEEE, 2009.

[16] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?,” in *International Conference on Software Engineering*, pp. 402–411, IEEE, 2005.

[17] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 654–665, ACM, 2014.

[18] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, “Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation,” *Software Engineering, IEEE Transactions on*, vol. 40, no. 1, pp. 23–42, 2014.

[19] P. Ammann and J. Offutt, *Introduction to Software Testing, 2nd edition*. Cambridge University Press, 2017.

[20] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” in *Proceedings of the 36th International Conference on Software Engineering*, pp. 435–445, ACM, 2014.

[21] “GitHub Repository for Mull.” `https://github.com/mull-project/mull`. [Online; accessed 19-October-2018].

[22] R. Just, B. Kurtz, and P. Ammann, “Inferring mutant utility from program context,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 284–294, ACM, 2017.

[23] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, “Mutant subsumption graphs,” in *Software testing, verification and validation workshops (ICSTW), 2014 IEEE seventh international conference on*, pp. 176–185, IEEE, 2014.

[24] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, “The care and feeding of wild-caught mutants,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 511–522, ACM, 2017.

[25] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, “Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults,” in *40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden*, 2018.

[26] M. Jimenez, T. Titcheu Chekam, M. Cordy, M. Papadakis, M. Kintis, Y. Le Traon, and M. Harman, “Are mutants really natural? a study on how naturalness helps mutant selection,” in *12th International Symposium on Empirical Software Engineering and Measurement (ESEM’18)*, 2018.

[27] X. Yao, M. Harman, and Y. Jia, “A study of equivalent and stubborn mutation operators using human analysis of equivalence,” in *Proceedings of the 36th International Conference on Software Engineering*, pp. 919–930, ACM, 2014.

[28] R. S. Freedman, "Testability of software components," *IEEE transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.

[29] J. M. Voas, "Pie: A dynamic failure-based technique," *IEEE Transactions on software Engineering*, vol. 18, no. 8, pp. 717–727, 1992.

[30] "JHawk." `http://www.virtualmachinery.com/jhawkprod.htm`. [Online; accessed 16-October-2018].

[31] R. E. Kalman, P. L. Falb, and M. A. Arbib, *Topics in mathematical system theory*, vol. 1. McGraw-Hill New York, 1969.

[32] M. Whalen, G. Gay, D. You, M. P. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 102–111, IEEE, 2013.

[33] R. Gopinath, C. Jensen, and A. Groce, "The theory of composite faults," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pp. 47–57, IEEE, 2017.

[34] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1100–1125, 2014.

[35] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[36] Q. Zhu, "GitHub Repository for Mutation Observer." `https://zenodo.org/badge/latestdoi/147203995`, 2019. [Online; accessed 18-January-2019].

[37] "Antlr." `http://www.antlr.org/`. [Online; accessed 29-October-2018].

[38] "Apache Commons BCEL." `https://commons.apache.org/proper/commons-bcel/`. [Online; accessed 29-October-2018].

[39] "java-callgraph GitHub Repositry." `https://github.com/gousiosg/java-callgraph`. [Online; accessed 29-October-2018].

[40] H. Coles, "PIT Mutation Operators." `http://pitest.org/quickstart/mutators/`. [Online; accessed 30-October-2018].

[41] D. E. Hinkle, W. Wiersma, S. G. Jurs, *et al.*, *Applied statistics for the behavioral sciences*. Houghton Mifflin Boston, 1988.

[42] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, Oct 2001.

[43] E. Frank, M. A. Hall, and I. H. Witten, *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, 4 ed., 2016.

[44] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[46] R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1993.

[47] M. R. Woodward, M. A. Hennell, and D. Hedley, "A measure of control flow complexity in program text," *IEEE Transactions on Software Engineering*, no. 1, pp. 45–50, 1979.

[48] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE transactions on software engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.

[49] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[50] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[51] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software evolution* (T. Mens and S. Demeyer, eds.), pp. 173–202, Springer, 2008.

[52] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. D. Lucia, “Automatic test case generation: What if test code quality matters?,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 130–141, ACM, 2016.

[53] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, “Analyzing the validity of selective mutation with dominator mutants,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 571–582, ACM, 2016.

[54] H. Coles, “Mutation testing - a practitioners perspective.” `https://github.com/hcoles/slides/blob/master/slides.pdf`. Accessed: 2017-05-09.

[55] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[56] R. A. DeMillo, “Test adequacy and program mutation,” in *Software Engineering, 1989. 11th International Conference on*, pp. 355–356, May 1989.

[57] A. J. Offutt and R. H. Untch, “Mutation 2000: Uniting the orthogonal,” in *Mutation testing for the new century*, pp. 34–44, Springer, 2001.

[58] T. T. Chekam, M. Papadakis, T. Bissyandé, Y. L. Traon, and K. Sen, “Selecting fault revealing mutants,” *arXiv preprint arXiv:1803.07901*, 2018.

[59] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, “A comprehensive study of pseudo-tested methods,” *Empirical Software Engineering*, pp. 1–31, 2017.