# How to Kill Them All

## An Exploratory Study on the Impact of Code Observability on Mutation Testing

Zhu, Qianqian ; Zaidman, Andy; Panichella, Annibale

**DOI**
10.1016/j.jss.2020.110864

**Publication date**
2021

**Document Version**
Accepted author manuscript

**Published in**
Journal of Systems and Software

**Citation (APA)**
Zhu, Q., Zaidman, A., & Panichella, A. (2021). How to Kill Them All: An Exploratory Study on the Impact of Code Observability on Mutation Testing. *Journal of Systems and Software*, *173*, 1-20. Article 110864. https://doi.org/10.1016/j.jss.2020.110864

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

Qianqian Zhu*, Andy Zaidman, Annibale Panichella

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

**Abstract**

Mutation testing is well-known for its efficacy to assess test quality, and starting to be applied in the industry. However, what should a developer do when confronted with a low mutation score? Should the test suite be plainly 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 aim to provide a new perspective to developers that enables them to understand and reason about the mutation score in the light of *testability* and *observability*. First, 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*, which measures the extent to which the production code is tested directly, 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 or adding tests to be able to improve the mutation score. In doing so, we observe that relatively simple refactoring operations enable an improvement or increase in the mutation score.

*Keywords:* mutation testing, code quality, observability, testability, code refactoring

## 1. Introduction

Mutation testing has been a very active research field since the 1970s as a technique to evaluate test suite quality in terms of the fault-revealing capability [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]. Further-

*Corresponding author

*Email addresses:* qianqian.zhu@hotmail.com (Qianqian Zhu), A.E.Zaidman@tudelft.nl (Andy Zaidman), A.Panichella@tudelft.nl (Annibale Panichella)

more, *Google* [5] has integrated mutation testing with the code review process for around 6000 software engineers.

As mutation testing gains traction in the industry, a better understanding of the *mutation score* (one outcome of mutation testing) becomes essential. The existing works have mainly linked the mutation score with test quality [6, 7] (i.e., *how good is the test suite at detecting faults in the software?*) and *mutant utility* [8, 9] (i.e., *how useful is the mutant?*). However, in our previous study, we have observed that certain mutants could be killed only after refactoring the production code to increase the *observability* of state changes. In such cases, test deficiency is not the only reason for the survival mutants. Still, some issues in the production code, such as *code observability*, result in difficulties to kill the mutants. Different from the previous works (e.g., [6, 7, 8, 9]), our goal is to bring a new perspective to developers that enable them to understand and reason about the mutation score in the light of *testability* and *observability*. Thereby, developers can make a choice when confronting low mutation scores: (1) adding new tests, (2) refactoring the production code to be able to write better tests, or (3) ignoring the surviving mutants.

To this aim, our study consists of two parts: firstly, we investigate the relationship between *testability/observability* and mutation testing in order to find the most correlated metrics; secondly, based on what we observe from the correlations, we define anti-patterns or indicators that software engineers can apply to their code to kill the surviving mutants. We start with investigating the relationship between *testability/observability metrics* and the mutation score inspired by the work of Bruntink and van Deursen [10]. *Testability* is defined as the "attributes of software that bear on the effort needed to validate the software product" [11, 10]. 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 [12]. Whalen et al. [13] 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 value, the expression is not *observable* for that test. Compared to testability that covers various aspects of a project (e.g., inheritance and cohesion), observability is more specifically addressing the extent to which the value change of expression is observable in a test case.

Our first three research questions steer our investigation in the first part of our study:

**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 help developers to take actions when facing survival mutants. That is why, based on the

observations from **RQ1-RQ3**, we define anti-patterns or indicators that software engineers can apply to their code/tests to ensure that mutants can be killed. This leads us to the next research question:

**RQ4** To what extent does the removal 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 with 16 code fragments to investigate whether the removal of anti-patterns increases the mutation score.

## 2. Background

In this section, we briefly introduce the basic concepts of and related works on mutation testing, testability metrics, and our proposed metrics for quantifying code observability.

### *2.1. Mutation Testing*

Mutation testing is defined by Jia and Harman [1] as a fault-based testing technique that 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) [14]. The benefits of mutation testing have been extensively investigated and can be summarised [15] as 1) having better fault exposing capability compared to other test coverage criteria [16, 17, 7], 2) being a valid substitute to real faults and providing a good indication of the fault detection ability of a test suite [18, 19].

Researchers have actively investigated mutation testing for decades (as evidenced by the extensive survey [14, 1, 20, 15]). 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 [21]. *Code coverage*, the most common metric to measure test suite effectiveness, has seen its limitations being reported in numerous studies (e.g. [16, 17, 7, 6]). Using structural coverage metrics alone might be misleading because, in many cases, statements might be covered, but their consequences might not be asserted [6]. Another factor is that well-developed open-source mutation testing tools (e.g., PIT/PiTest [3] and Mull [22]) have contributed to mutation testing being applied in the industrial environments [2, 5, 4].

However, questions still exist about mutation testing, especially regarding the usefulness of a mutant [9]. The majority of the mutants generated by existing mutation operators are equivalent, trivial, and redundant [23, 9, 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 important.

To address this knowledge gap, numerous studies have investigated how useful mutants are. Example studies include *mutant subsumption* [23], *stubborn* mutants [8], and *real-fault coupling* [19, 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 [10] (called software testability [27]), and since mutation testing can generally be considered as "testing the tests," production code quality could also impact mutation testing, just like production code quality has been shown to be correlated with the presence of test smells [28]. Due to the lack of insights into how code quality affects the efforts needed for mutation testing, especially in how to engineer tests that kill *all* the mutants, 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 [27] formally defined *observability* and *controllability* in the software domain. 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 understanding of *testability* by using statistical methods to predict *testability* based on various code metrics. Influential works include that of Bruntink and van Deursen [10], in which they explored the relationship between nine object-oriented metrics and testability. To explore the relation between *testability* and mutation score (**RQ1**), we first need to collect several existing object-oriented metrics that 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 select those 64 metrics because they measure various aspects of a project, including basic characteristics (e.g., NLOC and NOMT), inheritance (e.g., DIT), coupling (e.g., CBO and FIN), and cohesion (LCOM). A large number of those metrics, such as LCOM and HLTH, have been widely used to explore software testability [10, 30] and fault prediction [31, 32].

We present a brief summary of the 64 metrics in Table 1 (method-level) and Tables 2–3 (class-level). We computed these metrics using a static code analysis tool provided by `JHawk` [33].

### *2.3. Code Observability*

To explore the relation between *observability* and mutation score (**RQ2**), we first need a set of metrics to quantify *code observability*. According to Whalen et al. [13]'s definition of observability (as mentioned in Section 1), we consider that *code observability* comprises two perspectives: that of production code and

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 |

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` showing the method `setSectionPaint` and its corresponding test. This method is to set the section paint associated with the specified key for the `PiePlot` object, and sends a `PlotChangeEvent` to all registered listeners. There is one mutant in Line 3 that removes the call to `org/jfree/chart/plot/PiePlot ::fireChangeEvent`. This mutant is not killed by `testEquals`. Looking at the *observability* of this mutant from the *production code* perspective, we can see that the `setSectionPaint` 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 `testEquals` invokes the method `setSectionPaint` in Line 14 and 17, no proper assertion statements are used to examine the changes of `fireChangeEvent()` (which is used to send an event to listeners).

Starting with two angles of code observability, we come up with a set of the code observability metrics. Since our study is a starting point to design metrics

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 |

to measure the code observability, we start with the simple and practical metrics, which are easy for practitioners to understand and apply.

First of all, we consider the return type of the method. As discussed in Listing 1, it is hard to observe the changing states inside a void 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). The metric `is_void` is to examine whether the return value of the method is void or not. The metric `non_void_percent` addresses the return type at class level which measures the percent of non-void methods in the class. Besides these two, a void method might change 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 complement `is_void`.

Secondly, we come up with the access control modifiers. Let us consider

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 |
| cNLOC | Lines of Code | The number of lines of code in the class and its methods |

the example in Listing 2 from project `commons-lang-LANG_3_7`. 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 "`return if (getMantissa(str, str.length()) != null) null else throw new RuntimeException`"[1]. This mutant should be easy to detect given an input of either a legal String object (the return value is not null) or a null string (throw an exception). This "trivial" mutant is not detected because the method `getMantissa` is private. The access control modifier *private* makes it impossible to test the method `getMantissa` *directly*, for this method is only visible to methods from class `NumberUtils`. To test this method, the test case must first

[1]This mutant is generated by `Return Values Mutator` in PIT [34]. In Listing 2, getMantissa(str, str.length()) returns a `String` object. When the return value of a method is an object, the mutator replaces non-null return values with null and throw a `java.lang.RuntimeException` if the un-mutated method would return null.

```
public void setSectionPaint(Comparable key, Paint paint) {
  this.sectionPaintMap.put(key, paint);
  fireChangeEvent(); // mutant: remove this method
}

@Test
public void testEquals() {
...
  PiePlot plot1 = new PiePlot();
  PiePlot plot2 = new PiePlot();
  assertTrue(plot1.equals(plot2));
  assertTrue(plot2.equals(plot1));
  // sectionPaintMap
  plot1.setSectionPaint("A", new GradientPaint(1.0f, 2.0f,
    Color.BLUE,3.0f, 4.0f, Color.WHITE));
  assertFalse(plot1.equals(plot2));
  plot2.setSectionPaint("A", new GradientPaint(1.0f, 2.0f,
    Color.BLUE,3.0f, 4.0f, Color.WHITE));
  assertTrue(plot1.equals(plot2));
...
}
```

Listing 1: Example of method `org.jfree.chart.plot.PiePlot: setSectionPaint` and its test

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

Listing 2: Example of method `getMantissa` in class `NumberUtils`

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 code observability. Thereby, we take access control modifiers into account to quantify code observability, where we design `is_public` and `is_static` (shown in 4th and 5th rows in Table 5).

The third point we raise concerns fault masking. We have observed that mutants generated in *certain locations* are more likely to be *masked* [35], i.e., the state change cannot propagate to the output of the method. The first observation is that mutants that reside in a nested class. The reasoning is similar to mutants that reside in nested sections of code, namely that a change in intermediate results does not propagate to a point where a test can pick it up. Thus, we come up with `is_nested` (in 6th row in Table 5). Another group of mutants is generated inside nested conditions and loops. These can be problematic because the results of the mutations cannot propagate to the output, and the tests have no way of checking the intermediate results within the method. Accordingly, 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 through 13 rows in Table 5). The last observation is related to mutants that are inside a long method (the reason is similar to the mutants inside nested conditions and

```
@Override                                                                    1
public int hashCode() {                                                      2
  return (getLeft() == null ? 0 : getLeft().hashCode()) ^                    3
    (getMiddle() == null ? 0 : getMiddle().hashCode()) ^                     4
    (getRight() == null ? 0 : getRight().hashCode());                        5
}                                                                            6
```

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 |

loops), thus, we design `method_length` (shown in 14th row in Table 5).

The next aspect we consider is test directness. Before we dig into test directness, we take Listing 3 as an instance. Listing 3 shows the class `Triple` from project `commons-lang-LANG_3_7`, 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 test method directly invoking the method (production code) [36]. The direct test is useful because it allows to control the input data directly and to assert the output of a method directly. This example shows that test directness can influence the outcome of mutation testing, which denotes the test case angle of *code observability*. Previous works such as Huo and Clause [37] also addressed the significance of test directness in mutation testing. Therefore, we design two metrics, `direct_test_no.` and `test_distance` (shown in 15th and 16th row in Table 5), to quantify test directness. Those two metrics represent the *test case* perspective of code observability.

Last but not least, we take assertions into considerations. As discussed in Listing 1, we have observed that mutants without appropriate assertions in place (throwing exceptions is also under consideration) cannot be killed, as a prerequisite to killing a mutant is to have the tests fail in the mutated program. Schuler and Zeller [38] and Zhang and Mesbah [39] also drew a similar conclusion to ours. 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). The `assertion-McCabe_Ratio` metric [36] is originally proposed to measure *test completeness* by indicating the ratio between the number of the actual points of testing in the test code and the number of decision points in the production code (i.e., how many decision points are tested). For example, a method has a McCabe complexity of 4, then in the ideal case, we would expect 4 different assertions to test those linear independent paths (in this case this ration would be 1), but if the ratio is lower than 1, it could be an indication that either not all paths are tested, or that not all paths are tested in a direct way. The `assertion_density` metric [40] aims at measuring the ability of the test code to detect defects in the parts of the production code that it covers. We include those two metrics here as a way to measure the quality of assertions. These three metrics are proposed based on the *test case* perspective of code observability.

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

## 3. Experimental Setup

To examine our conjectures, we conduct an experiment using six open-source projects. We recall 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 removal of anti-patterns based on testability and observability help in improving the mutation score?*

### *3.1. Mutation Testing*

We adopt PIT (Version 1.4.0) [3] to apply mutation testing in our experiments. The mutation operators we adopt are the *default* mutation operators provided by PIT [34]: `Conditionals Boundary Mutator`, `Increments Mutator`, `Invert Negatives Mutator`, `Math Mutator`, `Negate Conditionals Mutator`, `Return Values Mutator`, and `Void Method Calls Mutator`. We did not adopt the extended set of mutation operators provided PIT, as the operators in the default version are largely designed to be stable (i.e., not be too easy to detect) and minimise the number of equivalent mutations that they generate [34].

Table 5: Summary of code 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 test methods directly invoking the method under test (production code)[2] | test directness |
| 16 | test_distance | The shortest method call sequence required to invoke the method (production code) by 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 its direct_test_no. is 0.
[3]If the method is directly tested, then its test_distance is 0. The maximum test_distance is set `Integer.MAX_VALUE` in Java which means there is no method call sequence that can reach the method from test methods.

### *3.2. Subject Systems*

We use six systems publicly available on GitHub in this experiment. Table 6 summarises the main characteristics of the selected projects, which include the lines of code (LOC), the number of tests (#Test), the total number of methods (#Total Methods), the number of selected methods used in our experiment (#Selected), the total number of mutants (#Total Mutants), and the killed mu-

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 |

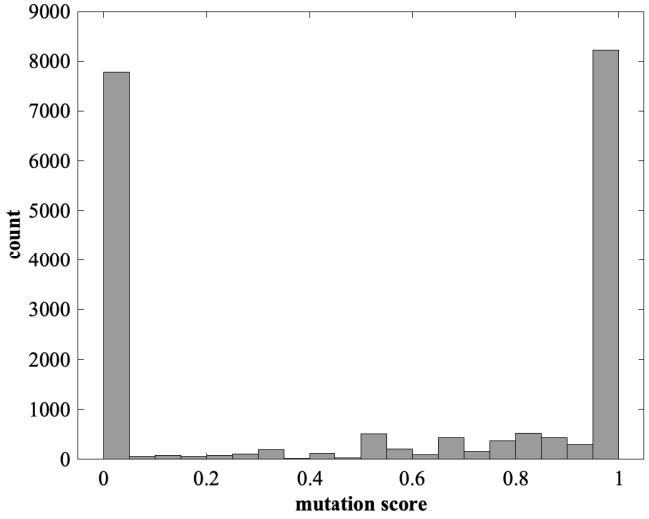
tants (#Killed). In our experiment, we remove the methods with no generated mutant by PIT, thus resulting in the number of selected methods (#Selected). These systems are selected because they have been widely used in the research domain (e.g., [38, 39, 37, 41, 42]). All systems are written in Java, and tested by means of JUnit. The granularity of our analysis is at the 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 selected methods. The majority of the mutation scores are either 0 or 1. Together with Figure 1b, we can see that the massive number of 0s and 1s are due to the low number of mutants per method. Most methods show less than 10 mutants, which is mainly due to most methods being short methods (`NOS` $< 2$ as shown in Figure 2). Writing short methods is a preferred strategy in practice, for a long method is a well-known code smells [43]. Besides, PIT adopts several optimisation mechanisms [44] to reduce the number of mutants. Thus, the number of mutants (#Total Mutants) shown in Table 6 is fewer than the actual number of generated mutants. The large number of the methods with low mutant number is an unavoidable bias in our experiment.

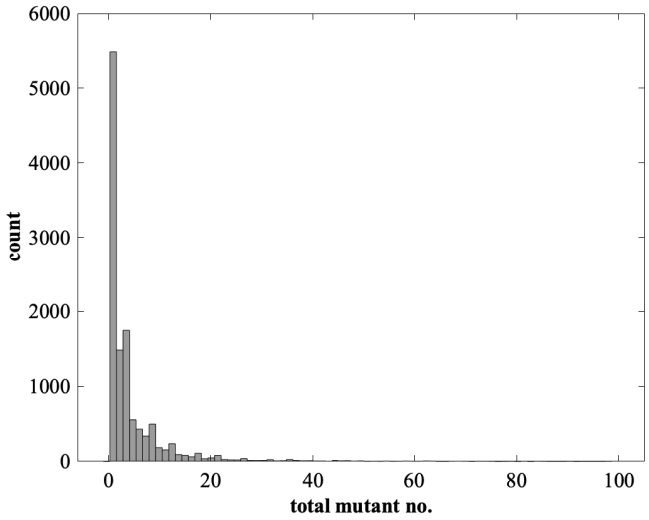
### *3.3. Tool implementation*

To evaluate the *code observability metrics* that we have proposed, we implemented 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 [45].

Our tool extracts information from three parts of the system under test (in Java): source code, bytecode, and tests. Firstly, Antlr [46] parses the source code to obtain the basic code features, e.g., **is public**, **is static**, and **(cond)**. Secondly, we adopt Apache Commons BCEL [47] to parse the bytecode. Then, `java-callgraph` [48] 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 PIT (Version 1.4.0) [3] to obtain the killable mutant results. An overview of the architecture of Mutation Observer can be seen in Figure 3.




(a) Distribution of mutation score per method




(b) Distribution of total mutant no. per method

Figure 1: Distribution of mutation score and mutant no.

### *3.4. Design of Experiment*

#### *3.4.1.* ***RQ1-RQ3***

Our investigation of the relationships between testability/observability metrics and the mutation score (**RQ1**-**RQ3**) is two-fold: in the first part, we adopt Spearman's rank-order correlation to measure the *pairwise correlations* statis-

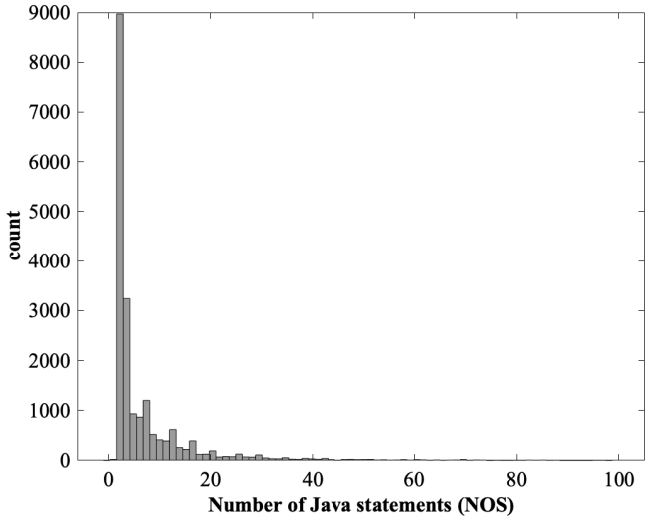



Figure 2: Distribution of Number of Java statements (NOS) per method

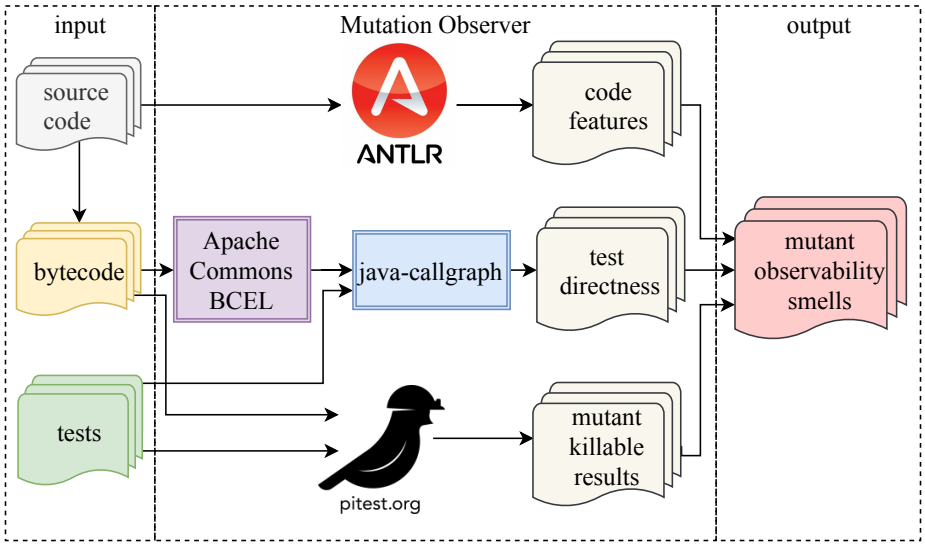



Figure 3: Overview of Mutation Observer architecture

tically between each metric (both testability and observability metrics) and the mutation score; in the second part, we turn the correlation problem into a binary classification problem (where we adopt Random Forest as the classification algorithm) to investigate how those metrics *interact* with one another.

***Pairwise correlations.*** To answer **RQ1**, **RQ2**, and **RQ3**, we first adopt Spearman's rank-order correlation to statistically measure the correlation between each metric (both testability and observability metrics) and the mutation score of the corresponding methods or classes. Spearman's correlation test checks whether there 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 into "negligible" ($|\rho| < 0.1$), "small" ($0.1 \leq |\rho| < 0.3$), "medium" ($0.3 \leq |\rho| < 0.5$), and "large" ($|\rho| \geq 0.5$) [49]. 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. To measure the statistical significance of Spearman's correlation test, we look at *p-values* that measure the probability of an observed (or more extreme) result assuming that the null hypothesis is true. Any test size larger than the p-value leads to rejection, whereas using a test size smaller than the p-value fails to reject the null hypothesis [50]. Here we consider the test size of 5% as the cutoff for statistical significance.

The mutation score[2] is calculated by Equation 1 (method-level).

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

We adopt `Matlab` [51] to calculate the Spearman's rank-order correlation coefficient between each metric and the mutation score. In particular, we used the statistical analysis (`corr` function with the option of "`Spearman`" in Matlab's default package[3]).

***Interactions.*** Except for the pairwise correlations between metrics and mutation score, we are also interested in how those metrics interact with one another. First, we try regression models to predict mutation scores based on the metrics. However, all the regression models incur extremely high cross-validation errors, i.e., Root Relative Squared Errors (RRSEs) are $> 70\%$ (e.g., RRSE of linear regression is 76.62%). Therefore, we turn the correlation problem into a classification problem for better performance. For simplicity, we use 0.5 as the cutoff between `HIGH` and `LOW` mutation core because 0.5 is widely used as a cutoff in classification problems whose independent variable ranges in [0,1] (e.g., defect prediction [52, 53]). We consider all the metrics to predicate whether

[2] In the original equation for mutation score, the divisor is the number of non-equivalent. In our study, our main focus is the relation between testability/observability metrics and mutation score, rather than mutation score itself. In our previous literature review [15], we have found that treating all mutants as non-equivalent is a common method when the mutation score is used as a relative comparison. Therefore, we do not manually analyse the equivalent mutants, and treated all mutants as non-equivalent.

[3] `https://www.mathworks.com/help/stats/corr.html`

the method belongs to classes with HIGH or LOW mutation score. One thing to notice here is that building a perfect prediction model is not our primary goal. Our interest is to see which metrics and/or which combinations of the metrics contributing to the LOW mutation score by building the prediction models. Therefore, deciding different threshold values is outside the scope of this paper.

For prediction, we adopt *Random Forest* [54] as the classification algorithm, where we use WEKA [55] 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 [56]. Besides, Random Forest is more accurate than one decision tree, and it is not affected by the overfitting problem [56].

As our investigation includes testability and observability metrics, for each project, we compare three types of classification models: (1) a model based on merely existing testability metrics, (2) a model based on merely code 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). In particular, we include the model based on the combination of the two aspects for further comparison: to see whether the combination of the two aspects can work better than each aspect itself. To examine the effectiveness of Random Forest 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 Forest model must thus perform better than ZeroR; otherwise, the Random Forest model is not suitable for our dataset.

In total, we consider four classification models: 1) ZeroR (i.e., the constant classifier), 2) Random Forest based on existing metrics, 3) Random Forest based on code observability metrics, and 4) Random Forest based on the combination of existing metrics and code observability metrics. To build Random Forest, WEKA [55] adopts bagging in tandem with random attribute selection. We use WEKA's default parameters to train the Random Forest model, i.e., "-P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1". To evaluate the performance of the classifier model (e.g., precision and recall), we use K-fold cross-validation with K=10 [57].

In terms of feature importance, we apply scikit-learn [58] to conduct the analysis. To determine the feature importance, scikit-learn [58] implements "*Gini Importance*" or "*Mean Decrease Impurity*" [59]. The importance of each feature is computed by the probability of reaching that node (which is approximated by the proportion of samples reaching that node) averaged over total tree ensembles [59]. We use the method of feature_importances_ in sklearn.ensemble.RandomForestRegressor [60] package to analyse the feature importance.

#### *3.4.2.* ***RQ4***

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 a low mutation score. To obtain the anti-pattern rules, we adopt J48 to build a decision tree [61, 55]. We consider J48 because of its advantage in interpretation over Random Forest. After building the decision tree, we rank all leaves (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.

### *3.5. Evaluation Metrics*

For **RQ1**, **RQ2**, and **RQ3**, to ease the comparisons of the four classification models, we consider four metrics widely used in classification problems: precision, recall, AUC, and the mean absolute error.

In our case, we cannot decide which class is positive or not, or in other words, we cannot say HIGH mutation score is what we expect. We use a prediction model to investigate the interactions between those metrics or how they interact with each other. So we adopt *weighted* precision and recall, which also take the number of instances in each class into consideration.

*Weighted precision.* The precision is the fraction of true positive instances in the instances that are predicted to be positive: TP/(TP+FP). The higher the precision, the fewer false positives. The *weighted precision* is computed as follows, where $p_{c1}$ and $p_{c2}$ are the precisions for class 1 and class 2, and $|c1|$ and $|c2|$ are the number of instances in class 1 and class 2, respectively:

$$weighted\,precision\ = \frac{p_{c1} \times |c1| + p_{c2} \times |c2|}{|c1| + |c2|} \tag{2}$$

*Weighted recall.* The recall is 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. The *weighted recall* is computed as follows, where $r_{c1}$ and $r_{c2}$ are the recalls for class 1 and class 2, and $|c1|$ and $|c2|$ are the number of instances in class 1 and class 2:

$$weighted\,recall\ = \frac{r_{c1} \times |c1| + r_{c2} \times |c2|}{|c1| + |c2|} \tag{3}$$

*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.

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 |

## 4. 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.

### *4.1. Spearman's rank order correlation*

#### *4.1.1. 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 metric 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 cutoff 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.

Overall, the pairwise correlation between each source code metric and the mutation score is not strong ($|rho| < 0.27$). 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 one another. For example, a long method does not necessarily have a low mutation score. Alternatively, another example: if there are more than four loops in a long method, then the method is very likely to have a low mutation score. That is also an example of the collinearity, i.e., the number of loops and the method length are highly correlated.

Table 8: Spearman results of code 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*[4] 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.

Additionally, 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. In fact, McCabe's Cyclomatic Complexity has been widely considered as 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 [62, 63, 64]). Based on our results without further investigation, we could only speculate that McCabe's Cyclomatic Complexity might not directly influence the mutation score.

***Summary***. *We found that the pair-wise correlations between the 64 existing source code metrics and the mutation score to be not so strong ($|rho| < 0.27$). The top 4 metrics with the strongest correlation coefficients are* `NSUP(class)`, `DIT(class)`, `R-R(class)` *and* `HIER(class)`.

*4.1.2. Observability*

***Findings***. Table 8 shows the overall results of Spearman's rank-order correlation analysis for code 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 code observability metrics and mutation score is still not strong (<0.5), but significantly better than existing code metrics (<0.27). The top five metrics are `test_distance`, `direct_test_no.`, `assertion-density`, `assertion-McCabe`, and `assertion_no`. The metrics related to *test directness*, i.e., `test_distance` (-0.4923) and `direct_test_no` (0.4177) are ranked first in

[4]In terms of absolute value.

Table 9: Random Forest Results of code observability metrics vs. Existing Metrics

| pid | ZeroR | | | | existing | | | | code observability | | | | combined | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | prec. | recall | AUC | err. | prec. | recall | AUC | err. | prec. | recall | AUC | err. | prec. | recall | AUC | err. |
| pid | prec. | recall | AUC | err. | prec. | recall | AUC | err. | prec. | recall | AUC | err. | prec. | recall | AUC | err. |
| 1 | - | 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 | 0.498 | 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 | 0.499 | 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 | 0.468 | 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 | 0.5 | 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 | 0.493 | 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 | 0.5 | 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 | 0.499 | 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 | 0.5 | 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 Importance 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 |

terms of *rho* among all metrics that we consider (including existing code metrics in Section 2.2). This observation corresponds to our hypothesis in Section 2.3 that the methods with no direct tests are more challenging to kill mutants. In terms of *rho* values, the assertion related metrics are ranked after test directness related metrics; this confirms both our conjectures in Section 2.3 and what has been reported in the related literature [38, 39] that the quality of assertions can influence the outcome of mutation testing.

***Summary***. *The correlations between code observability metrics and mutation score are not very strong (<0.5); however, they are 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`*).*

### *4.2. Random Forest*

***Classification effectiveness***. As discussed in Section 3.4, we compare the four models in terms of both our code observability metrics and the existing metrics, namely:

1. `ZeroR`: model using ZeroR approach

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 |

2. `existing`: Random Forest model based on existing code metrics
3. `code observability`: Random Forest model based on code observability metrics
4. `combined`: Random Forest model based on the combination of existing metrics and code observability metrics

Table 9 shows the results of fhe comparison of the four models. 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 the best performance is the one with the highest value, while for the mean absolute error, the best scoring model exhibits the lowest value. For the ZeroR model, because this model classifies all the instances to the majority (i.e., one class), the precision of the minority is not valid due to 0/0. Thus, in Table 9, we mark the precisions by "-".

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 code 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` (pid = 4). We conjecture that the number of instances (selected methods) in this project might be too small (only 150 methods) to develop a sound prediction model. In second place comes the model based on code 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 comes the `code observability`, slightly better than `existing`. Another interesting angle investigate further 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 importance 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 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 project; this is not surprising, as the task and context of these projects vary 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.

Looking at the overall dataset (Column "all" in Table 11), there are eight metrics from our proposed code 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 might have an impact on 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 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 4.1). 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 4.1). While in the feature importance analysis, assertion related metrics mostly rank after the top 5 (shown in Table 10 and Table 11) To further investigate the reason behind the dramatic changes

Table 12: Spearman Results of Test Directness vs. Assertions in terms of *rho*

| (rho) | assertion_no | assertion-McCabe | assertion_distance |
|---|---|---|---|
| direct_test_no | 0.9604 | 0.9472 | 0.9334 |
| test_distance | -0.8707 | -0.8707 | -0.8707 |

of ranks for assertion related metrics, we analyse the correlations between test directness (i.e., `direct_test_no` and `test_distance`) and assertion related metrics (i.e., `assertion_no`, `assertion-McCabe` and `assertion_distance`). Looking at the correlation results between test directness and assertion related metrics in Table 12, the major reason is that test directness and assertion related metrics are almost *collinear* in the prediction model (where $|rho| > 0.87$). To put simply, there are almost no tests without assertions for the six subjects. 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.

Moreover, we would like to put our observations into perspective by comparing our results with the work of Zhang et al. [42], where they have constructed a similar Random Forest model to predict the result of killable mutant based on a series of features related to mutants and tests. The metrics that are common to their model and ours are Cyclomatic Complexity (COMP), Depth of Inheritance Tree (DIT), `nested_depth`, Number of Subclasses (NSUB), and `method_length`. Only two metrics in their study, i.e., `method_length` (in 6th place) and `nested_depth` (in 10th place) appear in our top 15 (Column "all" in Table 11). Especially `COMP` which ranks nine in their results is not in our top 15. There are multiple reasons for the difference in results: (i) we do consider a much larger range of metrics, which provide a better explanatory power (statistically speaking) than the one in their paper; (ii) our goal is to determine patterns in production and test code that may prevent killing some mutants while Zhang et al. [42] predict if a mutant is killable (aka different prediction target and different granularity level). Besides, as we see later (next section), we can use our model to determine common anti-patterns with proper statistical methods. (iii) the subjects used in our experiment are different from theirs. For example, in project `java-apns-apns-0.2.3` (Column "4" in Table 11), `COMP` appears among the top 15.

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

## 5. RQ4 Code Refactoring

Our goal is to investigate whether we can refactor away the observability issue that we expect to hinder tests from killing mutants and thus to affect

Table 13: 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 |

Table 14: Top six anti-patterns from J48 decision tree 4

| Rule No. | Details |
|---|---|
| 1 | test_distance > 5 && (loop(loop)) $\leq$ 0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop) $\leq$ 0 && NOCL $\leq$ 9 && non-void_percent $\leq$ 0.42 |
| 2 | test_distance > 5 && (loop(loop)) $\leq$ 0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop) $\leq$ 0 && NOCL > 9 |
| 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 |
| 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 |
| 5 | test_distance $\leq$ 5 && is_void = 1 && nested_depth $\leq$ 0 && NOS $\leq$ 2 && assertion-density $\leq$ 0.14 && MOD > 1 |
| 6 | test_distance $\leq$ 5 && is_void = 1 && nested_depth $\leq$ 0 && NOS > 2 && assertion-density $\leq$ 0.22 && CREF > 1 && XMET > 0 |

the mutation score. In an in-depth case study, we manually analysed 16 code fragments to understand better the interaction between testability/observability 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)* [65] to perform feature selection, which, as Table 13 shows, leaves us with 36 features (or metrics). Then, as discussed in Section 3, we build a decision tree based on those 36 metrics using J48 (shown in Figure 4), and select the top 6 leaves (also called end nodes) in the decision tree for further *manual* analysis as potential refactoring guidelines. We present the top six anti-patterns in Table 14.

Here, we take a partial decision tree to demonstrate how we generate rules (shown in Figure 5). In Figure 5, we can see that there are three attributes (marked as an ellipse) and four end nodes or leaves (marked as a rectangle) in the decision tree. Since we would like to investigate how code refactoring increases 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 5). 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

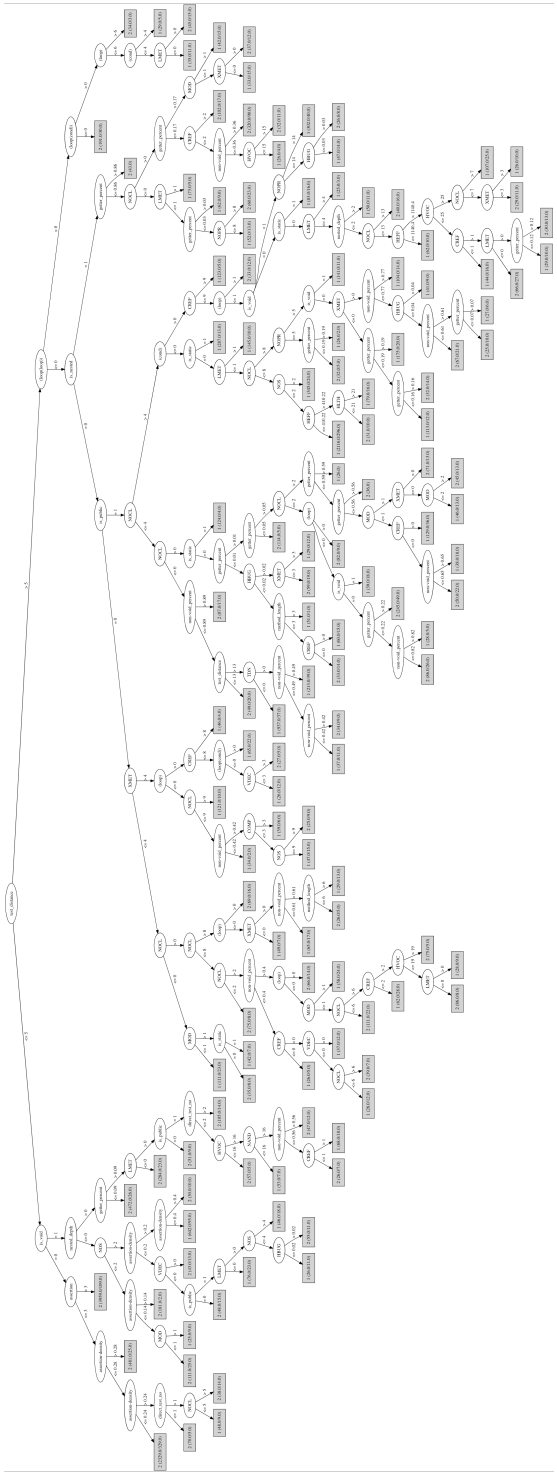


Figure 4: Overview of J48 decision tree

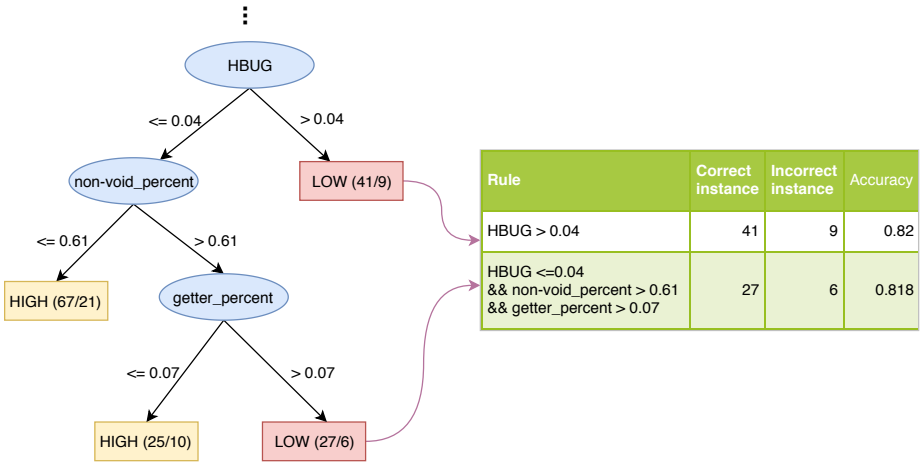



Figure 5: Demo of rule generation

Table 15: 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 |

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.

After selecting the rules, the first author of this paper has conducted the main task of the manual analysis. If there were any questions during the manual analysis, the attempts of refactoring or adding tests are discussed among all the authors to reach an agreement. In our *actual* case study, we manually analysed 16 cases in total. Due to space limitations, we only highlight six cases in this paper (all details are available on GitHub [45]). We will discuss our findings in code refactoring case by case.

### *5.1.* ***Case 1:*** `plot.MeterPlot::drawValueLabel` *from JFreeChart*

This case (shown in Listing 4) is under anti-pattern **Rule 1**: `test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop) ≤ 0 && NOCL ≤ 9 && non-void_percent ≤ 0.42`. In total, there are 5 mutants generated from this method (shown in Table 15). All 5 mutants survive the test suite.

```
/**                                                                    1139
 * Draws the value label just below the center of the dial.            1140
 *                                                                     1141
 * @param g2 the graphics device.                                      1142
 * @param area the plot area.                                          1143
 */                                                                    1144
protected void drawValueLabel(Graphics2D g2, Rectangle2D area) {       1145
  g2.setFont(this.valueFont);                                          1146
  g2.setPaint(this.valuePaint);                                        1147
  String valueStr = "No value";                                        1148
  if (this.dataset != null) {                                          1149
    Number n = this.dataset.getValue();                                1150
    if (n != null) {                                                   1151
      valueStr = this.tickLabelFormat.format(n.doubleValue()) + " "    1152
          + this.units;                                                1153
    }                                                                  1154
  }                                                                    1155
  float x = (float) area.getCenterX();                                 1156
  float y = (float) area.getCenterY() + DEFAULT_CIRCLE_SIZE;           1157
  TextUtils.drawAlignedString(valueStr, g2, x, y,TextAnchor.           1158
      TOP_CENTER);
}                                                                      1159
```

Listing 4: `plot.MeterPlot::drawValueLabel` (Case 1)

***Code changes.*** 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 the 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) to avoid the *assertion roulette*

```
@Test                                                                  1
public void testDrawValueLabel(){                                     2
  MeterPlot p1 = new MeterPlot(new DefaultValueDataset(1.23));         3
  BufferedImage image = new BufferedImage(3, 4, BufferedImage.         4
      TYPE_INT_ARGB);
  Graphics2D g2 = image.createGraphics();                              5
  Rectangle2D area = new Rectangle(0, 0, 1, 1);                        6
  p1.drawValueLabel(g2,area);                                          7
  assertTrue(g2.getFont() == p1.getValueFont());                       8
  assertTrue(g2.getPaint() == p1.getValuePaint());                     9
}                                                                      10
```

Listing 5: Direct test for Listing 4 (Case 1)

```
protected Rectangle2D drawValueLabel(Graphics2D g2, Rectangle2D area)    1145
    {
  g2.setFont(this.valueFont);                                            1146
  g2.setPaint(this.valuePaint);                                          1147
  String valueStr = "No value";                                          1148
  if (this.dataset != null) {                                            1149
    Number n = this.dataset.getValue();                                  1150
    if (n != null) {                                                     1151
      valueStr = this.tickLabelFormat.format(n.doubleValue()) + " "      1152
          + this.units;                                                  1153
    }                                                                    1154
  }                                                                      1155
  float x = (float) area.getCenterX();                                   1156
  float y = (float) area.getCenterY() + DEFAULT_CIRCLE_SIZE;             1157
  return TextUtils.drawAlignedString(valueStr, g2, x, y,TextAnchor.TOP_CENTER);  1158
}                                                                        1159
```

Listing 6: Refactoring of Listing 4 (Case 1)

Table 16: 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 |

test smell [66, 67]. By refactoring the method to non-void and adding a direct test, all previously surviving mutants are now successfully killed.

### *5.2.* ***Case 2:*** `axis.SymbolAxis::drawGridBands` *from JFreeChart*

This case (shown in Listing 8) is under **Rule 2**: `test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop) ≤ 0 && NOCL > 9`. In total, 4 mutants are generated from this method (see Table 16). None of the mutants are killed.

***Code changes.*** It is clear that this method is private, thus, it is impossible to call this method from outside the class directly. 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).

### *5.3.* ***Case 3:*** `builder.IDKey::hashCode` *from Apache Commons Lang*

This case (shown in Listing 11) is under **Rule 3**: `test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 1 && NOCL ≤ 4 && NOCL > 0 && is_static = 0 && getter_percent ≤ 0.01 && HBUG ≤ 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.

```
@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)

```
/**
 * 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: `axis.SymbolAxis::drawGridBands` (Case 2)

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

### *5.4.* ***Case 4:*** `AbstractCategoryItemRenderer::drawOutline` *from JFreeChart*

This case (shown in Listing 13) is under **Rule 4**: `test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 1 && NOCL > 4 && (cond) ≤ 0 && is_static = 0 && LMET ≤ 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 `AbstractCategoryPlot::drawOutline`. The original test suite did not kill the mutant.

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

```
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)

```
@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)

### 5.5. ***Case 5:*** `builder.ToStringStyle::setUseShortClassName` *from Apache Commons Lang*

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 `builder.ToStringStyle::setUseShortClassName`.

***Code changes.*** 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.

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

Listing 11: `builder.IDKey::hashCode` (Case 3)

```
@Test                                                                    1
public void testHashCode(){                                              2
  IDKey idKey = new IDKey(new Integer(123));                             3
  assertEquals(989794870,idKey.hashCode());                              4
}                                                                        5
```

Listing 12: Direct test for Listing 11 (Case 3)

```
/**                                                                      808
 * Draws an outline for the data area. The default implementation       809
     just
 * gets the plot to draw the outline, but some renderers will           810
     override this
 * behaviour.                                                           811
 *                                                                      812
 * @param g2 the graphics device.                                       813
 * @param plot the plot.                                                814
 * @param dataArea the data area.                                       815
 */                                                                     816
@Override                                                               817
public void drawOutline(Graphics2D g2, CategoryPlot plot,               818
    Rectangle2D dataArea) {                                             819
  plot.drawOutline(g2, dataArea);                                       820
}                                                                       821
```

Listing 13: `AbstractCategoryItemRenderer::drawOutline` (Case 4)

### 5.6. ***Case 6:*** `exception.TooManyEvaluationsException::<init>` *from Apache Commons Math*

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 `exception.util.ExceptionContext::addMessage`. This mutant is surviving the test suite.

***Code changes.*** 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` 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.

```
@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: `builder.ToStringStyle::setUseShortClassName` (Case 5)

### 5.7. *RQ4 Summary*

Based on all 16 cases that we analysed (available in our GitHub repository [45]), we found that our code observability metrics can lead to simple refactorings that enable to kill mutants that were previously not being killed. Ultimately, this leads to an increase 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`$\leq 5$.

```
@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)

```
/**                                                                    30
 * Construct the exception.                                           31
 *                                                                    32
 * @param max Maximum number of evaluations.                          33
 */                                                                   34
public TooManyEvaluationsException(Number max) {                      35
  super(max);                                                         36
  getContext().addMessage(LocalizedFormats.EVALUATIONS);              37
}                                                                     38
```

Listing 17: exception.TooManyEvaluationsException::<init> (Case 6)

```
public List<Localizable> getMsgPatterns(){                            1
  return msgPatterns;                                                 2
}                                                                     3
```

Listing 18: Refactoring of Listing 17 (Case 6)

- Private methods must be refactored to protected/public for testing (indicated by `is_public=0`).
- 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`).

### *5.8.* ***Discussion***

From the findings of **RQ4**, we can see that some code refactorings break OO design principles [68]. For instance, we suggest to change the access modifier from *private* to *protected/public* to kill the mutants; this violates the idea of Encapsulation, the ability to protect some components of the object from external entities [68]. A new *hypothesis* emerges from our study: the trade-off between OO design principles and testing and hence software testability [69]. The main

```
@Test                                                                 1
public void testMsgPatterns() {                                      2
  final int max = 12345;                                              3
  final TooManyEvaluationsException e = new                           4
      TooManyEvaluationsException(max);
  final String msg = e.getLocalizedMessage();                         5
  Assert.assertTrue(e.getContext().getMsgPatterns()                   6
        .contains(LocalizedFormats.EVALUATIONS));                     7
}                                                                     8
```

Listing 19: Additional assertion for Listing 17 (Case 6)

concepts of OO design are centred around the features of Data abstraction, Encapsulation, Inheritance, Polymorphism, and Dynamic binding. However, some factors such as Encapsulation and Inheritance could increase the complexity of OO systems and hence hinder testing and testability [69]. Existing literature [70, 71, 72, 73] has already addressed this dilemma. Mouchawrab et al. [70] pointed out that increasing the size of the inheritance hierarchy could increase the cost of testing due to dynamic dependencies. Singh and Saha's work [71] has shown that Inheritance and Polymorphism increase testing effort and lower software testability. All the works above indicate that there is a trade-off between OO design features and software testability. Currently, it is up to practitioners to balance the two perspectives themselves, depending on the requirements of software and their preferences.

In the context of mutation testing, a similar trade-off between OO design features and the ease of killing mutants exists. In this study, we relate the ease of killing mutants to the testability and observability. In Section 5.7, we found that a simple strategy to kill *all* the mutants is to write additional direct tests and/or assertions. However, some OO design features related to Encapsulation, such as the *private* access modifier (see Listing 8), increase the difficulty to add a direct test. Also, the void return type prevents killing the mutants generated from the immediate states that cannot propagate to the output (see Listing 4). As such, a very important note here is that our refactoring recommendations listed in Section 5.7 are centred around the anti-patterns based on the testability and observability; they *do not* take OO design principles into consideration. The recommendations attempt to help developers in understanding the cause of the low mutation score considering testability and observability, but not all surviving mutants are due to test quality.

Take Listing 8 for instance. The developer found the mutation score of this method is low, and our tool shows the low mutation score is mainly due to *private* access control modifier. Then, the developer can decide to ignore the surviving mutants if he cannot break Encapsulation based on the requirement. Or if this method is critical and must be well-tested according to the document, he may alter the access control modifier from *private* to *protected/public* to kill the mutants. Whether the developers make use of these testability and observability recommendations depends on their choices with regard to either (1) adding test cases [74, 75, 76] (2) refactoring the production code to kill the mutants, or (3) ignoring the surviving mutants.

## 6. Threats to Validity

*External validity.* Our results are based on mutants generated by the operators implemented in PIT. While PIT is a frequently used mutation testing tool, our results might be different when using other mutation tools [77]. Concerning the subject systems selection, we choose six open-source projects from GitHub; the selected projects differ in size, the number of test cases, and application domain. Besides, as mentioned in Section 3, the large number of methods with low number of mutants is an unavoidable bias in our experiment. The reason

is partly due to the optimisation mechanism of PIT [44] and partly due to a large number of short methods in those projects. Nevertheless, we do acknowledge that a broad replication of our study would mitigate any generalisability 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 rely on existing tools that have been widely used, e.g., `WEKA`, `MATLAB`, and PIT. 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 [78, 79]. Moreover, we consider equivalent mutants as a potential weakness in the software (reported by Coles [80, slide 44-52]); 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).

## 7. Related work

The notion of *software testability* dates back to 1991 when Freedman [27] 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. A prime example is the work of Bruntink and van Deursen [10], who have explored the relationship between nine class-level object-oriented metrics and testability. To the best of our knowledge, no study uses statistical or machine learning methods to investigate the relationship between *testability/observability* metrics and the 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 [81]. Since then, mutation 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 [82], Offutt and Untch [83], Jia and Harman [1], Offutt [14] and Zhu et al. [15]. Here we mainly address the studies that concern *mutant utility* [9], the efficacy of mutation testing. Yao et al. [8] have reported on the causes and prevalence of equivalent mutants and their relationship to stubborn mutants based on a manual analysis of 1230

mutants. Visser [84] has conducted an exhaustive analysis of all possible test inputs to determine how hard it is to kill a mutant considering three common mutation operators (i.e., relational, integer constants and arithmetic operators). His results show that *mutant reachability, mutation operators, and oracle sensitivity* are the key contributors to determining how hard it is to kill a mutant. Just et al. [9] 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. [85] 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 modeling techniques in mutation testing. All studies above 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 in terms of testability and observability affects the efforts needed for mutation testing, especially in how to engineer tests to kill *more* the mutants.

Similar to our study, there have been a few recent studies also investigating the relationships between assertions and test directness with mutation testing. Schuler and Zeller [38] introduced *checked coverage*—the ratio of statements that contribute to the computation of values that are later checked by the test suite—as an indicator for oracle quality. In their experiment, they compared checked coverage with the mutation score, where they found that checked coverage is more sensitive than mutation testing in evaluating oracle quality. Huo and Clause [37] proposed *direct coverage and indirect coverage* by leveraging the concepts of *test directness* with conventional statement coverage. They used the mutants as an indicator of the test suite effectiveness, and they found faults in indirectly covered code are significantly less likely to be detected than those in directly covered code. Zhang and Mesbah [39] evaluated the relationship between test suite effectiveness (in terms of the mutation score) and the (1) number of assertions, (2) assertion coverage, and (3) different types of assertions. They found test assertion quantity and assertion coverage are strongly correlated with the mutation score, and assertion types could also influence test suite effectiveness. Compared to our studies, those works only addressed one or two aspect(s) of *code observability* in our study. We provide a complete view of the relationships between code observability and mutation testing.

The study most related to ours is that of Zhang et al. [42]'s *predictive mutation testing*, where they have constructed a classification model to predict killable mutant 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 production code on mutation testing as their goal is to predict exact killable mutant results. An-

other interesting work close to our study is Vera-Pérez et al. [86]'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. [86]'s study and ours is that we pay attention to *conventional* mutation operators rather than "extreme mutation".

## 8. Conclusion & Future Work

This paper aims to bring a new perspective to software developers helping them to understand and reason about the mutation score in the light of *testability* and *observability*. This should enable developers to make decisions on the possible actions to take when confronted with low mutation scores. To achieve this goal, we firstly 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 *code observability metrics*, that are defined in terms of both production code and test cases, do show a stronger correlation with the mutation score ($|rho| < 0.5$). In particular, *test directness*, `test_distance`, and `direct_test_no` stand out.

To better understand the causality of our insights, we continue 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 and/or added tests according to the anti-patterns that we established in terms of the code observability metrics. Our aim here is to establish whether the removal of the observability anti-patterns would lead to an increase in the mutation score. We found that these anti-patterns can indeed provide insights in order to kill the mutants by indicating whether the production code or the test suite needs improvements. 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 purposes.

However, some refactoring recommendations could violate OO design principles. For example, by changing *private* to *protected/public* we increase observability, but we also break the idea of encapsulation. Therefore, we suggest developers make a choice between—(1) adding test cases, (2) refactoring the production code to kill the mutants, or (3) ignoring the surviving mutants—by considering the trade-off between OO design features and testability/observability.

To sum up, our paper makes the following contributions:

1. 19 newly proposed *code observability* metrics

2. A detailed investigation of the relationship between *testability/observability* metrics and the mutation score (**RQ1-RQ3**)
3. A case study with 16 code fragments to investigate whether removal of the anti-patterns increases the mutation score (**RQ4**)
4. A guideline for developers to make choices when confronting low mutation scores
5. A prototype tool coined MUTATION OBSERVER (openly available on GitHub [45]) that automatically calculates code observability metrics

***Future work.*** 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 code observability metrics by involving practitioners; 3) investigate the relations between more code metrics (e.g., code readability) and mutation score.


## Acknowledgements

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, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Software Eng. 37 (5) (2011) 649–678.

[2] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, 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), IEEE, 2018, pp. 47–53.

[3] H. Coles, GitHub Repository for PIT, `https://github.com/hcoles/pitest`, [Online; accessed 18-September-2019].

[4] H. Coles, PIT Main Page, `http://pitest.org/`, [Online; accessed 18-September-2019].

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

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

[7] N. Li, U. Praphamontripong, 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, IEEE, 2009, pp. 220–229.

[8] X. Yao, M. Harman, 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, ACM, 2014, pp. 919–930.

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

[10] M. Bruntink, A. van Deursen, An empirical study into class testability, Journal of systems and software 79 (9) (2006) 1219–1232.

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

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

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

[14] J. Offutt, A mutation carol: Past, present and future, Information and Software Technology 53 (10) (2011) 1098–1107.

[15] Q. Zhu, A. Panichella, A. Zaidman, A systematic literature review of how mutation testing supports quality assurance processes, Software Testing, Verification and Reliability 28 (6) (2018) e1675, e1675 stvr.1675. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1675, doi:10.1002/stvr.1675.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1675

[16] A. P. Mathur, W. E. Wong, An empirical comparison of data flow and mutation-based test adequacy criteria, Software Testing, Verification and Reliability 4 (1) (1994) 9–31.

[17] P. G. Frankl, S. N. Weiss, C. Hu, All-uses vs mutation testing: an experimental comparison of effectiveness, Journal of Systems and Software 38 (3) (1997) 235–253.

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

[19] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, 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, ACM, 2014, pp. 654–665.

[20] L. Madeyski, W. Orzeszyna, R. Torkar, M. Józala, Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation, Software Engineering, IEEE Transactions on 40 (1) (2014) 23–42.

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

[22] GitHub Repository for Mull, `https://github.com/mull-project/mull`, [Online; accessed 18-September-2019].

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

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

[25] M. Papadakis, D. Shin, S. Yoo, 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, 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] R. S. Freedman, Testability of software components, IEEE transactions on Software Engineering 17 (6) (1991) 553–564.

[28] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, A. Bacchelli, On the relation of test smells to software code quality, in: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 12–23.

[29] J. M. Voas, Pie: A dynamic failure-based technique, IEEE Transactions on software Engineering 18 (8) (1992) 717–727.

[30] J. Gao, M.-C. Shih, A component testability model for verification and measurement, in: 29th Annual International Computer Software and Applications Conference (COMPSAC'05), Vol. 2, IEEE, 2005, pp. 211–218.

[31] E. Arisholm, L. C. Briand, Predicting fault-prone components in a java legacy system, in: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, ACM, 2006, pp. 8–17.

[32] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2011) 1276–1304.

[33] JHawk, http://www.virtualmachinery.com/jhawkprod.htm, [Online; accessed 18-September-2019].

[34] H. Coles, PIT Mutation Operators, http://pitest.org/quickstart/mutators/, [Online; accessed 18-September-2019].

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

[36] D. Athanasiou, A. Nugroho, J. Visser, A. Zaidman, Test code quality and its relation to issue handling performance, IEEE Transactions on Software Engineering 40 (11) (2014) 1100–1125.

[37] C. Huo, J. Clause, Interpreting coverage information using direct and indirect coverage, in: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2016, pp. 234–243.

[38] D. Schuler, A. Zeller, Checked coverage: an indicator for oracle quality, Software testing, verification and reliability 23 (7) (2013) 531–551.

[39] Y. Zhang, A. Mesbah, Assertions are strongly correlated with test suite effectiveness, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 214–224.

[40] G. Kudrjavets, N. Nagappan, T. Ball, Assessing the relationship between software assertions and faults: An empirical investigation, in: 2006 17th International Symposium on Software Reliability Engineering, IEEE, 2006, pp. 204–212.

[41] Q. Zhu, A. Panichella, A. Zaidman, An investigation of compression techniques to speed up mutation testing, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2018, pp. 274–284.

[42] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, L. Zhang, Predictive mutation testing, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2018.2809496.

[43] K. Beck, M. Fowler, G. Beck, Bad smells in code, Refactoring: Improving the design of existing code (1999) 75–88.

[44] H. Coles, PIT Incremental Analysis, http://pitest.org/quickstart/incremental_analysis/, [Online; accessed 18-September-2019].

[45] Q. Zhu, GitHub Repository for Mutation Observer, https://zenodo.org/badge/latestdoi/147203995, [Online; accessed 18-September-2019] (2019).

[46] Antlr, http://www.antlr.org/, [Online; accessed 18-September-2019].

[47] Apache Commons BCEL, https://commons.apache.org/proper/commons-bcel/, [Online; accessed 18-September-2019].

[48] java-callgraph GitHub Repositry, https://github.com/gousiosg/java-callgraph, [Online; accessed 18-September-2019].

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

[50] H. J. Hung, R. T. O'Neill, P. Bauer, K. Kohne, The behavior of the p-value when the alternative hypothesis is true, Biometrics (1997) 11–22.

[51] MATLAB, version 9.6.0 (R2019a), The MathWorks Inc., Natick, Massachusetts, 2019.

[52] F. Zhang, Q. Zheng, Y. Zou, A. E. Hassan, Cross-project defect prediction using a connectivity-based unsupervised classifier, in: Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 309–320.

[53] A. Tosun, A. Bener, Reducing false alarms in software defect prediction by decision threshold optimization, in: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, 2009, pp. 477–480.

[54] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32. doi:10.1023/A:1010933404324.
URL https://doi.org/10.1023/A:1010933404324

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

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

[57] R. Kohavi, et al., A study of cross-validation and bootstrap for accuracy estimation and model selection, in: Ijcai, Vol. 14, Montreal, Canada, 1995, pp. 1137–1145.

[58] 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, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[59] L. Breiman, Classification and regression trees, Routledge, 2017.

[60] scikit-learn RandomForestRegressor, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html, [Online; accessed 18-September-2019].

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

[62] M. R. Woodward, M. A. Hennell, D. Hedley, A measure of control flow complexity in program text, IEEE Transactions on Software Engineering (1) (1979) 45–50.

[63] G. K. Gill, C. F. Kemerer, Cyclomatic complexity density and software maintenance productivity, IEEE transactions on software engineering 17 (12) (1991) 1284–1288.

[64] N. E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Transactions on Software engineering 26 (8) (2000) 797–814.

[65] S. Wold, K. Esbensen, P. Geladi, Principal component analysis, Chemometrics and intelligent laboratory systems 2 (1-3) (1987) 37–52.

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

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

[68] G. Booch, Object oriented analysis & design with application, Pearson Education India, 2006.

[69] P. R. Suri, H. Singhani, Object oriented software testability survey at designing and implementation phase, International Journal of Science and Research 4 (4) (2015) 3047–3053.

[70] S. Mouchawrab, L. C. Briand, Y. Labiche, A measurement framework for object-oriented software testability, Information and software technology 47 (15) (2005) 979–997.

[71] Y. Singh, A. Saha, Predicting testability of eclipse: a case study, Journal of Software Engineering 4 (2) (2010) 122–136.

[72] Y. Zhou, H. Leung, Q. Song, J. Zhao, H. Lu, L. Chen, B. Xu, An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems, Science china information sciences 55 (12) (2012) 2800–2815.

[73] M. Nazir, R. A. Khan, K. Mustafa, Testability estimation framework, International Journal of Computer Applications 2 (5) (2010) 9–14.

[74] M. Beller, G. Gousios, A. Zaidman, How (much) do developers test?, in: Proceedings of the International Conference on Software Engineering (ICSE - volume 2), IEEE, 2015, pp. 559–562.

[75] M. Beller, G. Gousios, A. Panichella, A. Zaidman, When, how and why developers (do not) test in their IDEs, in: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, 2015, pp. 179–190.

[76] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, A. Zaidman, Developer testing in the IDE: Patterns, beliefs, and behavior, IEEE Transactions on Software Engineering (TSE) 45 (3) (2019) 261–284.

[77] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, 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, ACM, 2016, pp. 571–582.

[78] B. J. Grün, D. Schuler, A. Zeller, The impact of equivalent mutants, in: 2009 International Conference on Software Testing, Verification, and Validation Workshops, IEEE, 2009, pp. 192–199.

[79] S. Mirshokraie, A. Mesbah, K. Pattabiraman, Efficient javascript mutation testing, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, IEEE, 2013, pp. 74–83.

[80] H. Coles, Mutation testing - a practitioners perspective, https://github.com/hcoles/slides/blob/master/slides.pdf, [Online; accessed 18-September-2019].

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

[82] R. A. DeMillo, Test adequacy and program mutation, in: Software Engineering, 1989. 11th International Conference on, 1989, pp. 355–356. doi:10.1109/ICSE.1989.714449.

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

[84] W. Visser, What makes killing a mutant hard, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 39–44.

[85] T. T. Chekam, M. Papadakis, T. Bissyandé, Y. L. Traon, K. Sen, Selecting fault revealing mutants, arXiv preprint arXiv:1803.07901 (2018).

[86] O. L. Vera-Pérez, B. Danglot, M. Monperrus, B. Baudry, A comprehensive study of pseudo-tested methods, Empirical Software Engineering (2017) 1–31.