**Noname manuscript No.**
(will be inserted by the editor)

# A Comprehensive Study of Pseudo-tested Methods

**Oscar Luis Vera-Pérez · Benjamin Danglot · Martin Monperrus · Benoit Baudry**

the date of receipt and acceptance should be inserted later



**Abstract** Pseudo-tested methods are defined as follows: they are covered by the test suite, yet no test case fails when the method body is removed, i.e., when all the effects of this method are suppressed. This intriguing concept was coined in 2016, by Niedermayr and colleagues, who showed that such methods are systematically present, even in well-tested projects with high statement coverage.

This work presents a novel analysis of pseudo-tested methods. First, we run a replication of Niedermayr's study with 28K+ methods, enhancing its external validity thanks to the use of new tools and new study subjects. Second, we perform a systematic characterization of these methods, both quantitatively and qualitatively with an extensive manual analysis of 101 pseudo-tested methods.

O. Vera-Pérez
Inria Rennes - Bretagne Atlantique
Campus de Beaulieu, 263 Avenue Général Leclerc
35042 Rennes - France
E-mail: oscar.vera-perez@inria.fr

B. Danglot
Inria Lille - Nord Europe Parc scientifique de la Haute Borne
40, avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq - France
E-mail: danglot@inria.fr

M. Monperrus
KTH Royal Institute of Technology in Stockholm
Brinellvägen 8
114 28 Stockholm - Sweden
E-mail: martin.monperrus@csc.kth.se

B. Baudry
KTH Royal Institute of Technology in Stockholm
Brinellvägen 8
114 28 Stockholm - Sweden
E-mail: baudry@kth.se

The first part of the study confirms Niedermayr's results: pseudo-tested methods exist in all our subjects. Our in-depth characterization of pseudo-tested methods leads to two key insights: pseudo-tested methods are significantly less tested than the other methods; yet, for most of them, the developers would not pay the testing price to fix this situation. This calls for future work on targeted test generation to specify those pseudo-tested methods without spending developer time.

## 1 Introduction

Niedermayr and colleagues [19] recently introduced the concept of *pseudo-tested methods*. These methods are covered by the test suite, but no test case fails even if all behaviors of the method are removed at once, i.e. when the body is completely stripped off. This work is novel and intriguing: such pseudo-tested methods are present in all projects, even those with test suites that have high coverage ratio.

If those results hold, it calls for more research on this topic. This is the motivation of this paper: first, we challenge the external validity of Niedermayr et al.'s experiment with new study subjects, second we perform an in-depth qualitative empirical study of pseudo-tested methods. In particular, we want to determine if pseudo-tested methods are indicators of badly tested code. While this seems to be intuitively true, we aim at quantifying this phenomenon. Second, we want to know whether pseudo-tested methods are relevant indicators for developers who wish to improve their test suite. In fact, these methods may encapsulate behaviors that are poorly specified by the test suite, but are not relevant functionalities for the project.

To investigate pseudo-tested methods, we perform an empirical study based on the analysis of 21 open source Java projects. In total, we analyze 28K+ methods in these projects. We articulate our study around three parts.

In the first part we characterize our study subjects by looking at the number and proportion of pseudo-tested methods. This also acts as a conceptual replication [22] of Niedermayr's study. Our results mitigate two threats to the validity of Niedermayr's results: our methodology mitigates internal threats, by using another tool to detect pseudo-tested methods, and our experiment mitigates external threats by augmenting Niedermayr's set of study objects with 17 additional open source project.

In the second part, we quantify the difference between pseudo-tested methods and the other covered methods. We compare both sets of methods with respect to the fault detection ratio of the test suite and prove that pseudo-tested methods are significantly worse tested with respect to this criterion.

In the third part, we aim at collecting a qualitative feedback from the developers. First, we manually found a set of pseudo-tested methods that reveal specific issues in the test suites of 7 projects. Then, we submitted pull requests and sent emails to the developers, asking their feedback about the relevance of these test issues. All pull requests have been merged to improve

the test suite. Second, we met with the developers of 3 projects and inspected together a sample of 101 pseudo-tested methods. We found that developers consider only 30 of them worth spending time improving the test suite.

To summarize, the contributions of this paper are as follow:

- a conceptual replication of Niedermayr's initial study on pseudo-tested methods. Our replication confirms the generalized presence of such methods, and improves the external validity of this empirical observation.
- a quantitative analysis of pseudo-tested methods, which measures how different they are compared to other covered, not pseudo-tested methods.
- a qualitative manual analysis of 101 pseudo-tested methods, involving developers, that reveals that less than 30% of these methods are clearly worth the additional testing effort.
- open science, with a complete replication package available at: `https://github.com/STAMP-project/descartes-experiments/`.

The rest of this paper is organized as follows. Section 2 defines the key concepts that form the core of this empirical study. Section 3 introduces the research questions that we investigate in this paper, as well as the set of study subjects and the metrics that we collect. In Section 4, we present and discuss the observations for each research question. In Section 5, we discuss the threats to the validity of this study and Section 6 discusses related works.

## 2 Pseudo-tested Methods

In this section, we first define the key concepts that we investigate in this study. Our definitions are founded on the presentation given by Niedermayr et al. [19]. Then, we describe the procedure that we elaborate, in order to automatically detect pseudo-tested methods.

### 2.1 Definitions

Here we define the main interactions between a program $P$ and its test suite $TS$, which form the basis of all dynamic analyses presented in this paper. From now on, we consider a program $P$ to be a set of methods.

**Definition 1** A test case $t$ is a method that initializes the program in a specific state, triggers specific behaviors and specifies the expected effects for these behaviors through assertions. $TS$ is the set of all test cases $t$ written for a program $P$.

**Definition 2** A method $m \in P$ is said to be covered if there exists at least one test case $t \in TS$ that triggers the execution of at least one path of the body of $m$.

Listing 1 illustrates an example of a test case according to Definition 1. This test case covers the `getInstance`, `getCharRanges`, and `toString` methods of the `CharSet` class. It initializes the `CharSet` object by calling `getInstance` and setting an initial state. Then, the `getCharRanges` method is invoked to obtain the character ranges composing the set specification. The three assertions in lines 7 - 9 specify the expected effects of the method invocations. Since at least one path is executed we say that `getInstance` is covered by `testConstructor` as stated in Definition 2. (Yet, not all the paths are covered, such as the instructions in lines 18 and 23.)

As per Definition 1, in a Java program, a set of a JUnit test classes is a particular case of a test suite.

```
1  class CharSetTest {
       ...
3      @Test
       public void testConstructor() {
5          CharSet set = CharSet.getInstance("a");
           CharRange[] array = set.getCharRanges();
7          assertEquals("[a]", set.toString());
           assertEquals(1, array.length);
9          assertEquals("a", array[0].toString());
       }
11     ...
   }
13
   class CharSet {
15     ...
       public static CharSet getInstance(final String... setStrs) {
17         if (setStrs == null) {
               return null;
19         }
           if (setStrs.length == 1) {
21             final CharSet common = COMMON.get(setStrs[0]);
               if (common != null) {
23                 return common;
               }
25         }
           return new CharSet(setStrs);
27     }
       ...
29     public CharRange[] getCharRanges() { ... }
   }
```

**Listing 1** A test case covering two methods of the class CharSet taken from `commons-lang`

Let $m$ be a method; $S = \cup_{m \in P} \mathit{effects}(m)$ the set of effects of all methods in $P$; $\mathit{effects}(m)$ a function $\mathit{effects} : P \to S$ that returns all the effects of a method $m$; *detect*, a predicate $TS \times S \to \{\top, \bot\}$ that determines if an effect is detected by $TS$. Here, we consider the following possible effects that a method can produce: change the state of the object on which it is called, change the state of other objects (by calling other methods), return a value as a result of its computation.

**Definition 3** A method is said to be pseudo-tested with respect to a test suite, if the test suite covers the method and does not assess any of its effects:

$$\forall s \in \mathit{effects}(m), \nexists t \in TS : \mathit{detect}(t, s)$$

**Definition 4** A method is said to be required if the test suite covers the method and assesses at least one of its effects:

$$\exists s \in \mathit{effects}(m), \exists t \in TS : \mathit{detect}(t, s)$$

```
class VList {
  private List elements;
  private int version;
  public void add(Object item) {
    elements.add(item);
    incrementVersion();
  }

  private void incrementVersion() {
    version++;
  }

  public int size() {
    return elements.size();
  }
}

class VListTest {
  @Test
  public void testAdd() {
    VList l = new VList();
    l.add(1);
    assertEquals(l.size(), 1);
  }
}
```

**Listing 2** Example of a pseudo-tested method

Listing 2 shows an example of a pseudo-tested method. `incrementVersion`, declared in line 9, is pseudo-tested. The test case in line 20 triggers the execution of the method but does not assess its effect: the modification of the `version` field. In the absence of other test cases, the body of this method could be removed and the test suite will not notice the change. This particular example also shows that pseudo-tested methods may pose a testing challenge derived from testability issues in the code.

One can note that Niedermayr et al. [19] call required methods, "tested methods". We do not keep this terminology here, for two key reasons: (i) these covered methods may include behaviors that are not specified by the test suite, hence not all the effects haven been tested; (ii) meanwhile, by contrast to pseudo-tested methods, the correct execution of these methods is *required* to make the whole test suite pass correctly since at least one effect of the method is assessed.

### 2.2 Tool for Finding Pseudo-tested Methods

A "pseudo-tested" method, as defined previously, is an idealized concept. In this section, we describe an algorithm that implements a practical way of collecting a set of pseudo-tested methods in a program $P$, in the context of the test suite $TS$, based on the original proposal of Niedermayr et al. [19]. It relies on the idea of "extreme code transformations", which consists in completely stripping out the body of a method.

Algorithm 1 starts by analyzing all methods of $P$ that are covered by the test suite and fulfill a predefined selection criterion (predicate ofInterest in line 1). This critetion is based on the structure of the method and aims at reducing the number of false positives detected by the procedure. It eliminates uninteresing methods such as trivial setter and getters or empty void methods. More insight on this will be given in Section 3.3. If the method returns a value, the body of the method is stripped out and we generate a few variants that simply return predefined values (line 3). These values depend on the return type, and are shown in Table 1, [1] If the method is void, we strip the body without further action (line 8). Once we have a set of variants, we run the test suite on each of them, if no test case fails on any of the variants of a given method, we consider the method as pseudo-tested (line 16). One can notice in line 13 that all extreme transformations are applied to the original program and are analyzed separately.

To conduct our study, we have implemented Algorithm 1 in an open source tool called Descartes[2]. The tool can detect pseudo-tested methods in Java programs tested with a JUnit test suite. Descartes is developed as an extension of PITest [3]. PITest is a state-of-the-art mutation testing tool for Java projects that works with all major build systems such as Ant Gradle and Maven. This mutation tool is under active maintenance and development. It provides several features for test running and selection and can be extended via plugins. Descartes leverages the maturity of PITest and handles the discovery of points where extreme transformations can be applied and the creation of the new program variants [25]. As a byproduct of the analysis and the features provided by PITest, Descartes is able to report the covered methods according to Definition 2. Being open-source, we hope that Descartes will be used by future research on the topic of pseudo-tested methods .

## 3 Experimental Protocol

Pseudo-tested methods are intriguing. They are covered by the test suite, their body can be drastically altered and yet the test suite does not notice the

[1] Compared to Niedermayr et al. [19], we add two new transformations, one to return *null* and another to return an empty array. These additions allow to expand the scope of methods to be analyzed.

[2] `https://github.com/STAMP-project/pitest-descartes`

```
   Data: P, TS
   Result: pseudo: {pseudo-tested methods in P}
 1 foreach m ∈ P|covered(m, TS) ∧ ofInterest(m) do
 2 |   variants : {extreme variants of m}
 3 |   if returnsValue(m) then
 4 |   |   stripBody(m)
 5 |   |   checkReturnType(m)
 6 |   |   variants ← fixReturnValues(m)
 7 |   end
 8 |   else
 9 |   |   variants ← stripBody(m)
10 |   end
11 |   failure ← false
12 |   foreach v ∈ variants do
13 |   |   P' ← replace(m, v, P)
14 |   |   failure ← failure ∨ run(TS, P')
15 |   end
16 |   if ¬failure then
17 |   |   pseudo ← pseudo ∪ m
18 |   end
19 end
20 return pseudo
```

**Algorithm 1:** Procedure to detect pseudo-tested methods

**Table 1** Extreme transformations used depending on return type.

| Method type | Values used |
|---|---|
| void | - |
| Reference types | null |
| boolean | true,false |
| byte,short,int,long | 0,1 |
| float,double | 0.0,0.1 |
| char | ' ', 'A' |
| String | "", "A" |
| T[] | new T[]{} |

transformations. We design an experimental protocol to explore the nature of those pseudo-tested methods.

### 3.1 Research Questions

Our work is organized around the following research questions:

- **RQ1** *How frequent are pseudo-tested methods?* This first question aims at characterizing the prevalence of pseudo-tested methods. It is a conceptual replication of the work by Niedermayr et al. [19], with a larger set of study objects and a different tool support for the detection of pseudo-tested methods.
- **RQ2** *Are pseudo-tested methods the weakest points in the program, with respect to the test suite?* This second question aims at determining to what

**Table 2** Projects used as study subjects. ▲: Projects taken from the work of Niedermayr et al. [19]. ▼: Projects taken from our industry partners. ♦: Other projects used in the software testing literature. ■: Project from Github with more than 12,000 commits. ★: Projects with more than a million LOC

| Project | ID | App LOC | Test LOC | Tests |
|---|---|---|---|---|
| ▼ AuthZForce PDP Core | `authzforce` | 12 596 | 3 463 | 634 |
| ★ Amazon Web Services SDK | `aws-sdk-java` | 1 676 098 | 24 115 | 1 291 |
| ♦ Apache Commons CLI | `commons-cli` | 2 764 | 4 241 | 460 |
| ♦ Apache Commons Codec | `commons-codec` | 6 485 | 10 782 | 663 |
| ▲ Apache Commons Collections | `commons-collections` | 23 713 | 27 919 | 13 677 |
| ♦ Apache Commons IO | `commons-io` | 8 839 | 15 495 | 963 |
| ▲ Apache Commons Lang | `commons-lang` | 23 496 | 37 237 | 2 358 |
| ♦ Apache Flink | `flink-core` | 46 390 | 30 049 | 2 341 |
| ♦ Google Gson | `gson` | 7 184 | 12 884 | 951 |
| ♦ Jaxen XPath Engine | `jaxen` | 12 467 | 8 579 | 716 |
| ▲ JFreeChart | `jfreechart` | 94 478 | 39 875 | 2 138 |
| ♦ Java Git | `jgit` | 75 893 | 52 981 | 2 760 |
| ♦ Joda-Time | `joda-time` | 28 724 | 55 311 | 4 207 |
| ♦ JOpt Simple | `jopt-simple` | 2 386 | 6 828 | 817 |
| ♦ jsoup | `jsoup` | 11 528 | 6 311 | 561 |
| ▼ SAT4J Core | `sat4j-core` | 18 310 | 8 091 | 710 |
| ♦ Apache PdfBox | `pdfbox` | 121 121 | 15 978 | 1 519 |
| ■ SCIFIO | `scifio` | 49 005 | 6 342 | 1 021 |
| ▼ Spoon | `spoon` | 48 363 | 32 833 | 1 371 |
| ▲ Urban Airship Client Library | `urbanairship` | 25 260 | 15 625 | 701 |
| ▼ XWiki Rendering Engine | `xwiki-rendering` | 37 571 | 9 276 | 2 247 |
| Total | | 2 332 671 | 424 215 | 42 106 |

extent pseudo-tested methods actually capture areas in the code that are less tested than other parts. Here we use the mutation score as a standard test quality assessment metric. We compare the method-level mutation score of pseudo-tested methods against that of other covered and required methods (that are not pseudo-tested). Here by method-level mutation score we mean the mutation score computed for each method, considering only the mutants created inside each particular method.

- **RQ3** *Are pseudo-tested methods helpful for developers to improve the quality of the test suite?* In this question we manually identify eight issues in the test suites of seven projects. We communicate these issues to the development teams through pull requests or email and collect their feedback.
- **RQ4** *Which pseudo-tested methods do developers consider worth an additional testing action?* Following our exchange with the developers, we expand the qualitative analysis to a sample of 101 pseudo-tested methods distributed across three of our study subjects. We consulted developers to characterize the pseudo-tested methods that are worth an additional testing action and the ones that are not worth it.

### 3.2 Study Subjects

We selected 21 open source projects in a systematic manner to conduct our experiments. We considered active projects written in Java, that use Maven as main build system, JUnit as the main testing framework and their code is available in a version control hosting service, mostly Github.

A project is selected if it meets one of the following conditions: 1) they are present in the experiment of Niedermayr et al. [19] (4 projects), 2) they are maintained by industry partners from whom we can get qualitative feedback (4 projects), 3) they are regularly used in the software testing literature (11 projects), 4) they have a mature history with more than 12,000 commits (one project) or they have a code base surpassing one million lines of code (one project).

Table 2 shows the entire list. The first two columns show the name of each project and the identifiers we use to distinguish them in the present study. The third and fourth columns of this table show the number of lines of code in the main code base and testing code respectively. Both numbers were obtained using *cloc*[3]. The last column shows the number of test cases as reported by Maven when running *mvn test*. For instance, Apache Commons IO (`commons-io`) has 8 839 lines of application code and 3 463 lines of test code spread over 634 test cases. The smallest project, Apache Commons Cli (`commons-cli`), has 2 764 lines, while the largest, Amazon Web Services (`aws-sdk-java`) is composed of 1.6 million lines of Java code. The list shows that our inclusion criteria enable us to have a wide diversity in terms of project size. In many cases the test code is as large or larger than the application code base.

We have prepared a replication package at:
`https://github.com/STAMP-project/descartes-experiments/`.

### 3.3 Metrics

In this section, we define the metrics we used to perform the quantitative analysis of pseudo-tested methods.

**Number of methods (#METH).** The total number of methods found in a project, after excluding constructors and static initializers. We make this choice because these methods cannot be targeted by our extreme transformations.

Certain types of methods are not generally targeted by developers in unit tests, we exclude them to reduce the number of methods that developers may consider as false positives. In our analysis, we do not consider:

- methods that are not covered by the test suite. We ignore these methods, since, by definition, pseudo-tested methods are covered.
- `hashCode` methods, as suggested by Niedermayr et al. [19], since this type of transformation would still convey with the hash code protocol

[3] `https://github.com/AlDanial/cloc`

– methods with the structure of a simple getter or setter (i.e. methods that only return the value of an instance field or assign a given value to an instance field), methods marked as deprecated (i.e. methods explicitly marked with the (@Deprecated) annotation or declared in a class marked with that same annotation), empty void methods, methods returning a literal constant and compiler generated methods such as synthetic methods or methods generated to support *enum* types

**Number of methods under analysis (#MUA).** Given a program $P$, that includes #METH methods, the number of methods under analysis #MUA is obtained after excluding the methods described above.

**Ratio of covered methods (C_RATE).** A program $P$ can be seen as a set of methods. When running a test suite $TS$ on $P$ a subset of methods $COV \subset P$ are covered by the test suite. The ratio of covered methods is defined as C_RATE= $\frac{|COV|}{|P|}$. In practice, $COV$ is computed by our tool, as explained in Section 2.2.

**Ratio of pseudo-tested methods (PS_RATE).** For all methods under analysis, we run the procedure described in Algorithm 1, to get the subset of pseudo-tested methods, noted as #PSEUDO methods. The ratio of pseudo-tested methods of a program is defined as PS_RATE=$\frac{\#\text{PSEUDO}}{\#\text{MUA}}$.

PS_RATE is used in RQ1 to determine the presence of pseudo-tested methods in our study subjects. We also use the Pearson coefficient to check if we can state a correlation between PS_RATE and C_RATE.

**Mutation score**. Given a program $P$, its test suite $TS$ and a set of mutation operators *op*, a mutation tool generates a set $M$ of mutants for $P$. The mutation score of the test suite $TS$ over $M$ is defined as the ratio of detected mutants included in $M$. That is:

$$score(M) = \frac{|\mu : \mu \in M \wedge detected(\mu)|}{|M|} \quad (1)$$

where $detected(\mu)$ means that at least one test case in $TS$ fails when the suite is executed against $\mu$.

**Mutation score for pseudo-tested methods (MS_pseudo)**: the score computed with the subset of mutants generated by applying the mutation operators only on pseudo-tested methods.

**Mutation score for required methods (MS_req)**: the score computed with the subset of mutants generated by applying the mutation operators only on required methods.

These three mutation score metrics are used in RQ2 to quantitatively determine if pseudo-tested methods are the worst tested methods in the code base. We perform a Wilcoxon statistical test to compare the values obtained for MS_pseudo and MS_req on our study subjects.[4]

[4] The computation of the Pearson coefficient and the Wilcoxon test were performed using the features of the `R` language.

## 4 Experimental Results

The following sections present in depth our experimental results.

### 4.1 RQ1: How frequent are pseudo-tested methods?

We analyzed each study subject following the procedure described in Section 2.2. The results are summarized in Table 3. The second column shows the total number of methods excluding constructors. The third, lists the methods covered by the test suite. The following column shows the ratio of covered methods. The "#MUA" column shows the number of methods under analysis, per the criteria described in Section 3.3. The last two columns give the number of pseudo-tested methods (#PSEUDO) and their ratio to the methods under analysis (PS_RATE).

For instance, in `authzforce`, 325 methods are covered by the test suite, of which 291 are relevant for the analysis. In total, we identify 13 pseudo-tested methods representing 4% of the methods under analysis.

We discover pseudo-tested methods in all our study objects, even for those with high coverage. This corroborates the observations made by Niedermayr et al. [19]. The number of observed pseudo-tested methods ranges from 2 methods, in `commons-cli`, to 473 methods in `jfreechart`. The PS_RATE varies from 1% to 46%. In 14 cases its value remains below 7% of all analyzed methods. This means that, compared to the total number of methods in a project, the amount of pseudo-tested methods can be managed by the developers in order to guide the improvement of test suites.

#### *4.1.1 Analysis of outliers*

Some projects have specific features that may affect the PS_RATE value in a distinctive way. In this section we discuss those cases.

`authzforce`, uses almost exclusively parameterized tests. The ratio of covered methods is low, but these methods have been tested exhaustively and therefore they have a lower PS_RATE compared to other projects with similar coverage.

The application domain can have an impact on the design of test suites. For example, `scifio` is a framework that provides input/output functionalities for image formats that are typically used in scientific research. This project shows the highest PS_RATE in our study. When we look into the details, we find that 62 out of their 72 pseudo-tested methods belong to the same class and deal with the insertion of metadata values in DICOM images, a format widely used in medical imaging. Not all the metadata values are always required and the test cases covering these methods do not check their presence. `pdfbox` is another interesting example in this sense. It is a library designed for the creation and manipulation of PDF files. Some of their functionalities can only be checked by visual means which increases the level of difficulty to specify

an automated and fine-grained oracle. Consequently, this project has a high PS_RATE.

At the other end of the PS_RATE spectrum, we find `commons-cli` and `jopt-simple`. These are small projects, similar in purpose and both have comprehensive test suites that reach 97% and 98% of line coverage respectively (as measured by *cobertura*[5]). Only two pseudo-tested methods were found for each one of them. Three of those four methods create and return an exception message. The remaining method is a `toString` implementation.

#### *4.1.2 Relationship between pseudo-tested methods and coverage*

We observe that the projects with lowest method coverage show higher ratios of pseudo-tested methods. The Pearson coefficient between the coverage ratio and the ratio of pseudo-tested methods is -0.67 and $p < 0.01$ which indicates a moderate negative relationship.

This confirms our intuition that pseudo-tested methods are more frequent in projects that are poorly tested (high pseudo-tested ratios and low coverage ratios). However, the ratio of pseudo-tested methods is more directly impacted by the way the methods are verified and not the ratio of methods covered. It is possible to achieve a low ratio of pseudo-tested methods covering a small portion of the code. For example, `authzforce` and `xwiki-rendering` have comparable coverage ratios but the former has a lower ratio of pseudo-tested methods. The correlation with the ratio is a consequence of the fact that, in general, well tested projects also have higher coverage ratios.

#### *4.1.3 Comparison with the study by Niedermayr et al. [19]*

Our study, on a new dataset, confirms the major finding of Niedermayr et al. [19]'s study: pseudo-tested methods exist in all projects, even the very well tested ones. This first-ever replication improves the external validity of this finding. We note in the original study by Niedermayr et al. [19], that the reported ratio was higher, ranging from 6% to 53%. The difference can be explained from the fact that 1) we exclude deprecated methods and 2) we consider two new other mutation operators. These two factors change the set of methods that have been targeted.

> We have made the first independent replication of Niedermayr et al. [19]'s study. Our replication confirms that all Java projects contain pseudo-tested methods, even the very well tested ones. This improves the external validity of this empirical fact. The ratio of pseudo-tested methods with respect to analyzed methods ranged from 1% to 46% in our dataset.

[5] `http://cobertura.github.io/cobertura/`

**Table 3** Number of methods in each project, number of methods under analysis and number of pseudo-tested methods

| Project | #Methods | #Covered | C_RATE | #MUA | #PSEUDO | PS_RATE |
|---|---|---|---|---|---|---|
| `authzforce` | 697 | 325 | 47% | 291 | 13 | 4% |
| `aws-sdk-java` | 177 449 | 2 314 | 1% | 1 800 | 224 | 12% |
| `commons-cli` | 237 | 181 | 76% | 141 | 2 | 1% |
| `commons-codec` | 536 | 449 | 84% | 426 | 12 | 3% |
| `commons-collections` | 2 729 | 1 270 | 47% | 1 232 | 40 | 3% |
| `commons-io` | 875 | 664 | 76% | 641 | 29 | 5% |
| `commons-lang` | 2 421 | 1 939 | 80% | 1 889 | 47 | 2% |
| `flink-core` | 4 133 | 1 886 | 46% | 1 814 | 100 | 6% |
| `gson` | 624 | 499 | 80% | 477 | 10 | 2% |
| `jaxen` | 958 | 616 | 64% | 569 | 11 | 2% |
| `jfreechart` | 7 289 | 3 639 | 50% | 3 496 | 476 | 14% |
| `jgit` | 6 137 | 3 702 | 60% | 2 539 | 296 | 12% |
| `joda-time` | 3 374 | 2 783 | 82% | 2 526 | 82 | 3% |
| `jopt-simple` | 298 | 265 | 89% | 256 | 2 | 1% |
| `jsoup` | 1 110 | 844 | 76% | 751 | 28 | 4% |
| `sat4j-core` | 2 218 | 613 | 28% | 585 | 143 | 24% |
| `pdfbox` | 8 164 | 2 418 | 30% | 2 241 | 473 | 21% |
| `scifio` | 3 269 | 895 | 27% | 158 | 72 | 46% |
| `spoon` | 4 470 | 2 976 | 67% | 2 938 | 213 | 7% |
| `urbanairship` | 2 933 | 2 140 | 73% | 1 989 | 28 | 1% |
| `xwiki-rendering` | 5 002 | 2 232 | 45% | 2 049 | 239 | 12% |
| Total | 234 923 | 32 650 | 14% | 28 808 | 2 540 | 9% |

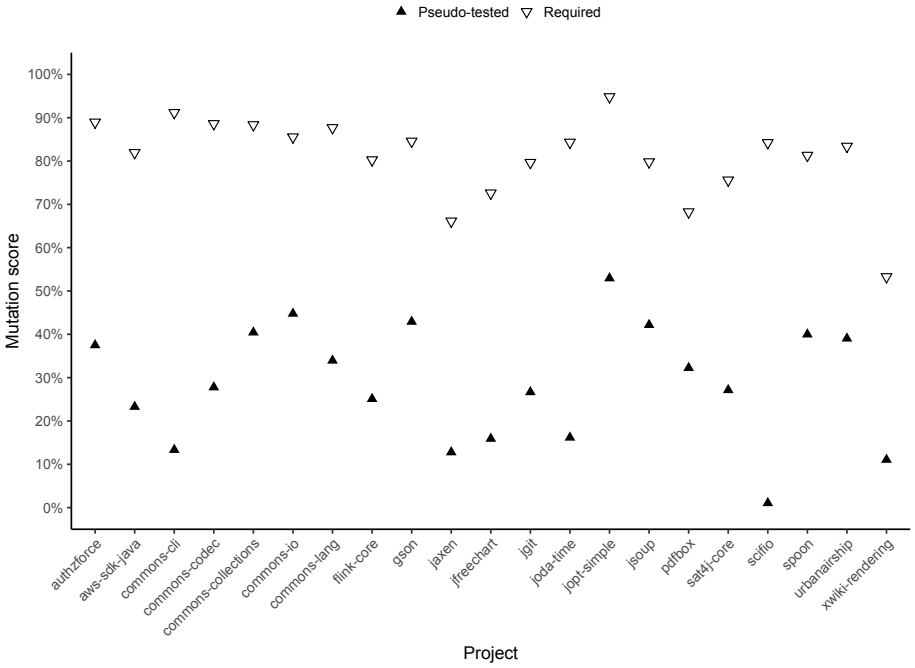
### 4.2 RQ2: Are pseudo-tested methods the weakest points in the program, with respect to the test suite?

By definition, test suites fail to assess the presence of any effect in pseudo-tested methods. As such, these methods can be considered as very badly tested, even though they are covered by the test suite. To further confirm this fact we assess the test quality of these methods with a traditional test adequacy criterion: mutation testing [7]. To do so, we measure the chance for a mutant planted in a pseudo-tested method to be detected (killed).

For each of our study subjects, we run a mutation analysis based on PITest, a state of the art mutation tool for Java. We configure PITest with its standard set of mutation operators. PITest is capable of listing: the comprehensive set of mutants, the method in which they have been inserted and whether they have been detected (killed) by the test suite. We extract the set of mutants that have been placed in the body of the pseudo-tested methods to compute the mutation score on those methods (MS_pseudo) as well as the mutation score of required methods (MS_req).

Figure 1 shows the results of this experiment. In all cases, the mutation score of pseudo-tested methods is significantly lower than the score of normal required methods. This means that a mutant planted inside a pseudo-tested method has more chances to survive than a mutant planted in required methods. The minimum gap is achieved in `pdfbox` with scores 32% for pseudo-tested methods and 68% for others. For `scifio`, only 1% of PITest mutants in

**Fig. 1** Mutation score for mutants placed inside pseudo-tested and required methods.




pseudo-tested methods can be killed (as opposed to 84% in required methods). To validate this graphical finding, we compare MS_pseudo and MS_req.

In average the mutation score of required methods is 52% above that of pseudo-tested methods. With the Wilcoxon statistical test, this is a significant evidence of a difference with a p-value $p < 0.01$. The effect size is 1.5 which is considered as large per the standard guidelines in software engineering [16].

### *4.2.1 Analysis of interesting examples*

It calls the attention, that, in no case, MS_pseudo was 0%. So, even when extreme transformations are not spotted by the test suite, some mutants inside these methods can be detected. We now explain this case.

Listing 3 shows a simplified extract of a pseudo-tested method we have found in `auzthforce` and where some traditional mutants were detected. The `checkNumberOfArgs` method is covered by six test cases and was found to be pseudo-tested. In all test cases, the value of `numInputs` is greater than two, hence the condition on line 3 was always false and the exception was never thrown. PITest created five mutants in the body of this method and two of them were detected. Those mutants replaced the condition by *true* and $<$ by $<=$ respectively. With this, the condition is always $false$, the exception is thrown and the mutants are detected. It means that those mutants are trivially

detected with an exception, not by an assertion. This is the major reason for which the mutation score of pseudo-tested methods can be higher than 0.

This explanation holds for `jopt-simple` which achieves a seemingly high 53% MS_pseudo. A total of 17 mutants are generated in the two pseudo-tested methods of the project. Nine of these mutants are killed by the test suite. From these nine, six replaced internal method calls by a default value and the other three replaced a constant by a different value. All nine mutations made the program crash with an exception, and are thus trivially detected.

```
class AnyOfAny {
  protected void checkNumberOfArgs(int numInputs) {
    if(numInputs < 2)
      throw new IllegalArgumentException();
  }

  public void evaluate(... args) {
    checkNumberOfArgs(args.size())
    ...
  }
}
```

**Listing 3** A pseudo-tested method where traditional mutants were detected

`scifio` has the lowest MS_pseudo. PITest generated 7 598 mutants in the 62 methods dealing with metadata and mentioned in Section 4.1.1. The mutants modify the metadata values to be placed in the image and, as discussed earlier, those values are not specified in any oracle of the test suite. Hence, none of these mutants are detected.

### *4.2.2 Distribution of method-level mutation score*

To further explore the difference between MS_pseudo and MS_req, we compute the distribution of the method-level mutation score. That is, we compute a mutation score for each method in a project. The final distribution for each project is shown in Figure 2. Each row displays two violin plots for a specific project.[6] Each curve in the plot represents the distribution of mutation scores computed per method. The thicker areas on each curve represent scores achieved by more methods.

The main finding is that the distributions for the required methods are skewed to the right (high mutation score), while the scores for pseudo-tested methods tend to be left skewed. This is a clear trend, which confirms the results of Figure 1. It is also the case that most distributions cover a wide range of values. While most pseudo-tested methods have low scores, there are cases for which the mutation score could reach high values, due to trivial exception-raising mutants.

In these plots the already discussed phenomenon for the methods of `scifio` also becomes visible. Those methods have each a considerable number of mu-

[6] The violin plot for pseudo-tested methods of `commons-cli` and `jopt-simple` are not displayed, as they have too few methods in this category.

tants and none of them were detected, therefore the `scifio` distribution for pseudo-tested methods is remarkably left skewed.

We observe that 63 pseudo-tested methods across all projects have a 100% mutation score. Among those 63, 34 have only one or two trivial mutants. As an extreme case, in the `jsoup` project we find that the method `load` of the `Entities` class[7] is pseudo-tested and PITest generates 69 mutants that are all killed. All mutants make the program crash with an exception, yet the body of the method can be removed and the absence of effects is unnoticed by the test suite. This suggests that the extreme transformations performed to find pseudo-tested methods are less susceptible to be trivially detected.

> The hypothesis that pseudo-tested methods expose weakly tested regions of code is confirmed by mutation analysis. For all the 21 considered projects, the mutation score of pseudo-tested methods is significantly lower than the score of required methods, a finding confirmed by a very low p-value lower than 0.01 and a very high effect size of 1.5.

## 4.3 RQ3: Are pseudo-tested methods relevant for developers to improve the quality of the test suite?

To answer this question, we manually analyze the *void* and *boolean* pseudo-tested methods which are accessible from an existing test class. *void* and *boolean* methods have only one or two possible extreme transformations, thus are easier to explain to developers. We identify eight testing issues revealed by these pseudo-tested methods: two cases of a miss-placed oracle, two cases of missing oracle, three cases of a weak oracle and one case of a missing test input. These issues have been found in seven of our study subjects.

For each testing issue we prepare a pull request that fixes the issue, or we send the information by email. Our objective is to collect qualitative feedback from the development teams about the relevance of the testing issues revealed by pseudo-tested methods.

We now summarize the discussion about each testing issue.

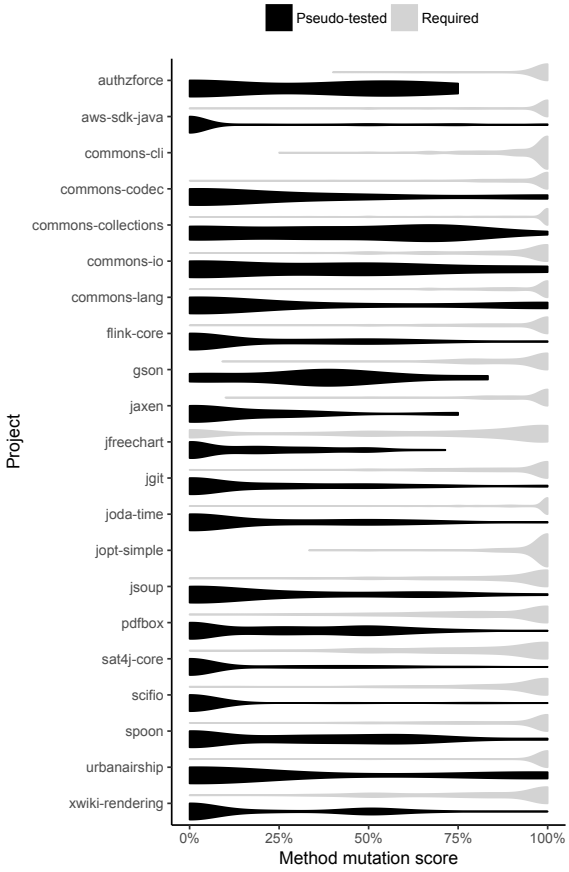
### *4.3.1 Feedback from `aws-sdk-java`*

Per our selection criterion, we have spotted one pseudo-tested method. We made one pull request (PR)[8] to explicitly assess the effects of one pseudo-tested method named `prepareSocket`. This method is covered by four test cases that follow the same pattern. A simplified extract is shown in Listing 4. The test cases mock a socket abstraction that verifies if a given array matches the expected value. `prepareSocket` should call the `setEnabledProtocols` in the

[7] `https://github.com/jhy/jsoup/blob/35e80a779b7908ddcd41a6a7df5f21b30bf999d2/src/main/java/org/jsoup/nodes/Entities.java#L295`

[8] `https://github.com/aws/aws-sdk-java/pull/1437`

**Fig. 2** PITest method-level mutation score distribution by project and method category




socket abstraction. When running an extreme transformation on this method, the assertion is never evaluated and the test cases pass silently. In the pull request we moved the assertion out of the `setEnabledProtocols` method, in order to have it verified after `prepareSocket`. Listing 5 shows a simplified version of the proposed code. With this modification, the method is not pseudo-tested anymore. The developer agreed that the proposed change was an improvement and the pull request was merged into the code. This is an example of a miss-placed oracle and the value of pseudo-tested methods.

```
1  @Test
   void typical() {
3    SdkTLSSocketFactory f = ...;
     //prepareSocket was found to be pseudo-tested
5    f.prepareSocket(new TestSSLSocket() {
       ...
7      @Override
       public void setEnabledProtocols(String[] protocols) {
```

```
9       assertTrue(Arrays.equals(protocols, expected));
        }
11    ...
      });
13  }
```

**Listing 4** A weak test case for method prepareSocket.

```
1   @Test
    void typical() {
3     SdkTLSSocketFactory f = ...;
      SSLSocket s = new TestSSlSocket() {
5     @Override
      public void setEnabledProtocols(String[] protocols) {
7       capturedProtocols = protocols;
      }
9     ...
      };
11    f.prepareSocket(s);
      //This way the test fails if no protocol was enabled
13    assertArrayEquals(s.capturedProtocols, expected));
    }
```

**Listing 5** Proposed test improvement. The assertion was moved out of the socket implementation. Consequently, `prepareSocket` is no longer pseudo-tested

### *4.3.2 Feedback from `commons-collections`*

In this project, certain methods implementing iterator operations are found to be pseudo-tested. Specifically two implementations of the `add` method and four of the `remove` method are pseudo-tested in classes where these operations are not supported. Listing 6 shows one the methods and a simplified extract of the test case designed to assess their effects. If the `add` method is emptied, then the exception is never thrown and the test passes. We proposed a pull request [9] with the change shown in Listing 7. The proposed change verifies that an exception has been thrown. As in the previous example, the issue is related to the placement of the assertion. The developer agreed to merge the proposed test improvement into the code. This is a second example of the value of pseudo-tested methods. Being in a different project, and assessed by another developer, this increases our external validity.

```
    class SingletonListIterator
2     implements Iterator<Node> {
      ...
4     void add() {
      //This method was found to be pseudo-tested
6     throw new UnsupportedOperationException();
      }
8     ...
    }
10
```

[9] `https://github.com/apache/commons-collections/pull/36`

```
  class SingletonListIteratorTest {
12    ...
      @Test
14    void testAdd() {
      SingletonListIterator it = ...;
16    ...
      try {
18      //If the method is emptied, then nothing happens
        //and the test passes.
20      it.add(value);
      } catch(Exception ex) {}
22    ...
  }
```

**Listing 6** Class containing the pseudo-tested method and the covering test class.

```
1 ...
  try {
3   it.add(value);
    fail(); //If this is executed,
5       //then the test case fails
  } catch(Exception ex) {}
7 ...
```

**Listing 7** Change proposed in the pull request to verify the unsupported operation.

### *4.3.3 Feedback from* `commons-codec`

For `commons-codec` we found that the *boolean* method `isEncodeEqual` was pseudo-tested. The method is covered by only one test case, shown in Listing 8. As one can notice, the test case lacks the corresponding assertion. So, none of the extreme transformations applied to this method could cause the test to fail.

```
1 public void testIsEncodeEquals() {
    final String[][] data = {
3   {"Meyer", "M\u00fcller"},
    {"Meyer", "Mayr"},
5   ...
    {"Miyagi", "Miyako"}
7 };
    for (final String[] element : data) {
9           final boolean encodeEqual =
    this.getStringEncoder().isEncodeEqual(element[1], element[0]);
11 }
}
```

**Listing 8** Covering test case with no assertion.

All the inputs in the test case should make the method return *true*. When we placed the corresponding assertion we found that the first input (in line 3) was wrong and we replaced it by a correct pair of values. We made a pull

request[10] and the fixture was accepted by the developers and also slightly increased the code coverage by 0.2%.

### *4.3.4 Feedback from* `commons-io`

In `commons-io` we found several *void* `write` methods of the `TeeOutputStream` to be pseudo-tested. This class represents an output stream that should send the data being written to other two output streams. A reduced version of the test case covering these methods can be seen in Listing 9. Line 7 shows that the assertion checks that both output streams should contain the same data. If the `write` method is emptied, nothing is written to both streams but the assertion remains valid as both have the same content (both are empty). The test case should verify not only that those two streams have the same content but that they have the right value. In this sense, we say that this is an example of a weak oracle. We made a pull request[11] with the changes exposed in Listing 10. The change adds a third output stream to be used as a reference value. The pull request was accepted and it slightly increased the code coverage by 0.07%.

```
public void testTee() {
  ByteArrayOutputStream baos1 = new ByteArrayOutputStream();
  ByteArrayOutputStream baos2 = new ByteArrayOutputStream();
  TeeOutputStream tos = new TeeOutputStream(baos1, baos2);
  ...
  tos.write(array);
  assertByteArrayEquals(baos1.toByteArray(), baos2.toByteArray());
}
```

**Listing 9** Test case verifying TeeOutputStream write methods.

```
public void testTee() {
  ByteArrayOutputStream baos1 = new ByteArrayOutputStream();
  ByteArrayOutputStream baos2 = new ByteArrayOutputStream();
  ByteArrayOutputStream expected = new ByteArrayOutputStream();
  TeeOutputStream tos = new TeeOutputStream(baos1, baos2);
  ...
  tos.write(array);
  expected.write(array);
  assertByteArrayEquals(expected.toByteArray(), baos1.toByteArray
      ());
  assertByteArrayEquals(expected.toByteArray(), baos2.toByteArray
      ());
}
```

**Listing 10** Change proposed to verify the result of the write methods

[10] https://github.com/apache/commons-codec/pull/13
[11] https://github.com/apache/commons-io/pull/61

For three projects, `spoon`, `flink-core` and `sat4j-core`, we discuss the details of the testing issues[12] directly with the developers via emails. We systematically collected their feedback.

### *4.3.5 Feedback from `spoon`*

We ask the project team about a public *void* method, named `visitCtAssert`[13], and covered indirectly by only one test case. This method was part of a visitor pattern implementation, which is common inside this project. This particular method handles *assert* Java expressions in an Abstract Syntax Tree. The test case does not assess the effects of the method. The developers expressed that this method should be verified by adding a stronger verification or a new test case. They were interested in our findings. They took no immediate action but opened a general issue [14].

### *4.3.6 Feedback from `flink-core`*

This team was contacted to discuss about a public *void* method, named `configure`[15], which is directly called by a single test case. This particular method loads a given configuration and prevents a field value from being overwritten. Listing 11 shows a simplified extract of the code. The body of the method could be removed and the test passes as the assertion only involves the initial value of the field. The developers explained that the test case was designed precisely to verify that the field is not changed after the method invocation. They expressed that more tests could probably make the scenario more complete. In our view, the test case should assert both facts: the configuration being loaded and the value not being changed. The current oracle expresses a weaker condition as the former verification is not done. If the body is erased, the configuration is never loaded and the value, of course, is never changed. The developers did not take any further action.

```
1 public void configure(Configuration parameters) {
    super.configure(parameters);
3   if(this.blockSize == NATIVE_BLOCK_SIZE) {
      setBlockSize(...);
5   }
  }
```

**Listing 11** Pseudo-tested method in `flink-core`

[12] https://github.com/STAMP-project/descartes-experiments/blob/6f8a9c7c111a1da5794622652eae5327d0571ef1/direct-communications.md

[13] https://github.com/INRIA/spoon/blob/fd878bc71b73fc1da82356eaa6578f760c70f0de/src/main/java/spoon/reflect/visitor/DefaultJavaPrettyPrinter.java#L479

[14] https://github.com/INRIA/spoon/issues/1818

[15] https://github.com/apache/flink/blob/740f711c4ec9c4b7cdefd01c9f64857c345a68a1/flink-core/src/main/java/org/apache/flink/api/common/io/BinaryInputFormat.java#L86

#### *4.3.7 Feedback from* `sat4j-core`

We contacted the `sat4j-core` lead developer about two *void* methods. One of them, named `removeConstr`[16], was covered directly by only one test case to target a specific bug and avoid regression issues. The other method, named `learn`[17], was covered indirectly by 68 different test cases. The lead developer considered the first method as helpful to realize that more assertions were needed in the covering test case. Consequently, he made one commit[18] to verify the behavior of this method.

The second pseudo-tested method was considered a bigger problem, because it implements certain key optimizations for better performance. The tests cases triggered the optimization code but did not leverage the optimized result. Their result were the same with or without the optimization code. Consequently, the developer made a new commit[19] with an additional, more complex, test case where the advantages of the optimization could be witnessed.

#### *4.3.8 Discussion*

We now discuss the main findings of this qualitative user study.

First, all developers agreed that it is easy to understand the problems identified by pseudo-tested methods. This confirms the fact that, we, as outsiders to those projects, with no knowledge or experience, can also grasp the issue and propose a solution. The developers acknowledged the relevance of the uncovered flaws.

Second, when developers were given the solution for free (through pull requests written by us), they accepted the test improvement.

Third, when the developers were only given the problem, they did not always act by improving the test suite. They considered that pseudo-tested methods provide relevant information, and that it would make sense to enhance their test suites to tackle the issues. But they do not consider these improvements as a priority. With limited resources, the efforts are directed to the development of new features and to fix existing bugs, not to improve existing tests.

Of the eight testing issues found, seven can be linked to oracle issues and one to an input problem.

[16] `https://gitlab.ow2.org/sat4j/sat4j/blob/09e9173e400ea6c1794354ca54c36607c53391ff/org.sat4j.core/src/main/java/org/sat4j/tools/xplain/Xplain.java#L214`
[17] `https://gitlab.ow2.org/sat4j/sat4j/blob/09e9173e400ea6c1794354ca54c36607c53391ff/org.sat4j.core/src/main/java/org/sat4j/minisat/core/Solver.java#L384`
[18] `https://gitlab.ow2.org/sat4j/sat4j/commit/afab137a4c1a54219f3990713b4647ff84b8bfea`
[19] `https://gitlab.ow2.org/sat4j/sat4j/commit/46291e4d15a654477bd17b0ce905926d24e042ca`

Pseudo-tested methods uncover flaws in the test suite which are considered relevant by developers. These methods enable one to well understand the problem in a short time. However, fixing the test flaws requires some time and effort that cannot always be given, due to higher priority tasks such as new features and bug fixing.

### 4.4 RQ4: Which pseudo-tested methods do developers consider worth an additional testing action?

To answer this question we contact the development teams directly. We select three projects for which the developers have accepted to discuss with us: `authzforce`, `sat4j-core` and `spoon`. We set up a video call with the head of each development team. The goal of the call is to present and discuss a selection of pseudo-tested methods in approximately 90 minutes. With this discussion, we seek to know which pseudo-tested methods developers consider relevant enough to trigger additional work on the test suite and approximate their ratio on each project.

For projects `sat4j-core` and `spoon`, we randomly choose $\tilde{2}5\%$ of all pseudo-tested methods. The third project, `authzforce`, has only 13 of such methods so we consider them all, as it is a number that can be discussed in reasonable time. We prepared a report for the developers that contains the list of pseudo-tested methods, with the extreme transformations that were applied and the test cases covering the method. To facilitate the discussion we also included links to the exact version of the code we analyzed. This information was made available to the developers before the meeting.

For each method, we asked the developers to determine if: 1) given the structure of the method, and, 2) given its role in the code base, they consider it is worth spending time creating new test cases or fixing existing ones to specify those methods. We also asked them to explain the reasons behind their decision.

Table 4 shows the projects involved, footnotes with links to the online summary of the interviews, the number of pseudo-tested methods included in the random sample, the number of methods worth an additional testing action and the percentage they represent with respect to the sample. We also show how much time we spent in the discussion.

We observe that only 23% of pseudo-tested methods in `sat4j-core` and `spoon`, the two largest projects, are worth additional testing actions (having the same percentage is purely coincidental). For `authzforce` the percentage

[20] `https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/authzforce-core/sample.md`

[21] `https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/sat4j-core/sample.md`

[22] `https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/spoon/sample.md`

**Table 4** The pseudo-tested methods systematically analyzed by the lead developers, through a video call.

| Project | Sample size | Worth | Percentage | Time spent (HH:MM) |
|---|---|---|---|---|
| `authzforce`[20] | 13 (100%) | 6 | 46% | 29 min |
| `sat4j-core`[21] | 35 (25%) | 8 | 23% | 1 hr 38 min |
| `spoon`[22] | 53 (25%) | 16 | 23% | 1 hr 14 min |
| Total | 101 | 30 | 30% | 3 hr 21 min |

of methods to be specified is 46%, but the absolute number (6) does not differ much for `sat4j-core` (8). This indicates that, potentially, many pseudo-tested come from functionalities considered less important or not a priority, therefore not well tested. The proportion of methods considered as worthless additional testing appears surprisingly high. It is important to notice that, among pseudo-tested methods, developers find cases, in their own words, "surprising" and "definitively not well tested, but they should be". Even for the cases they don't consider important, a developer from `sat4j-core` state that they "would like to know in which scenarios the transformation was discovered".

We now enumerate the main reasons given by developers to consider a method worth or worthless spending time creating specific testing actions. We also include the projects in which these reason manifested.

**Worthless specifying:** A pseudo-tested method could be considered as useless to test, i.e., *not important*, if it meets one of the following criteria:

- The code has been automatically generated (`spoon`).
- The method is part of debug functionalities, i.e., formatting a debug or log message or creating and returning an exception message or an exception object (`authzforce`).
- The method is part of features that are not widely used in the code base or in external client code (`authzforce`, `sat4j-core`, `spoon`).
- The method has not been deprecated but its functionality is being migrated to another interface (`spoon`).
- The code of the method is considered as simple or trivial to need a specific test case (`sat4j-core`, `spoon`). Short methods, involving a few simple instructions are generally not considered worth to be specified by a direct unit test cases (`spoon`). Listing 13 shows two examples that `spoon` developers consider too simple to to be worth of additional testing actions.
- Methods created just to complete an interface implementation (`spoon`). The object oriented design may involve classes that need to implement a given interface but do not actually need to provide a behavior for all methods. In those cases, developers write a placeholder body which they are not interested in testing.
- Receiving methods in a delegation pattern that add little or no logic when invoking the delegate (`spoon`). The delegation pattern exposes a method that simply calls another method (delegate). Delegate methods may have the same signature as the receiving method. The receiving method usually

```
...
public void addArrayReference(CtArrayTypeReference<?>
    typeReference) {
    arrayTypeReference.setComponentType(typeReference);
}
...
```

**Listing 12** Pseudo-tested method involving a delegation pattern.

```
...
public void externalState() {
  this.selectedState = external;
}
...

public boolean matches(CtElement e) {
  e.setFactory(f);
  return false;
}
```

**Listing 13** Pseudo-tested methods considered as too simple to require more testing actions.

adds no or very little custom logic (e.g., provide a default value for unused parameters or process the returning value). Listing 12 shows an example of this pattern that developers do not consider to be worth of additional testing actions. If the delegate is pseudo-tested then the receiving method will be pseudo-tested as well. The opposite does not have to be necessarily true. In any case, the method exposing the actual functionality to be tested is the delegate. The receiving method may not have the same importance.

**Worth specifying with additional tests:** On the other hand, developers provided the following reasons when they consider a pseudo-tested method to be worth of additional testing actions:

- A method that supports a core functionality of the project or part of the main responsibility of the declaring class (`authzforce`, `sat4j-core`, `spoon`). For example, we find a class named `VisitorPartialEvaluator` which implements a visitor pattern over a Java program Abstract Syntax Tree (AST). This class simplifies the AST by evaluating all expressions that could be statically reduced. The method `visitCtAssignment`[23], declared in this class, handles assignment instructions and was found to be pseudo-tested. Assignments may influence the evaluation result, so this method plays an important role in the class.
- A method supporting a functionality that is widely used in the code base. It could be the method itself that is being frequently used or the class that declares the method (`authzforce`, `sat4j-core`, `spoon`).

[23] `https://github.com/INRIA/spoon/blob/fd878bc71b73fc1da82356eaa6578f760c70f0de/src/main/java/spoon/support/reflect/eval/VisitorPartialEvaluator.java#L515`

```
protected final void checkNumberOfArgs(final int numInputs)
{
  if (numInputs != 3)
  {
    throw new IllegalArgumentException(...);
  }
}
```

**Listing 14** A simple method that checks a precondition

- A method known to be relevant for external client code (`sat4j-core`).
- A new feature that is partially supported or not completed yet, which requires a clear specification (`spoon`).
- Methods which are the only possible way to access certain features of a class (`authzforce`). For example, a public method that calls several private methods which actually contain the implementation of the public behavior.
- Method verifying preconditions (`authzforce`). These methods guarantee the integrity of the operations to be performed. Listing 14 shows an example of one of those methods considered to be important to specify. Despite the simplicity of the implementation, the `authzforce` developers consider that it is important to specify them as accurately as possible.

We have observed cases where a method meets criteria to be worth of specification and at the same time to be worthless of additional testing actions. The final decision of developers in those cases is subjective and responds to their internal knowledge about the code base. This means that it is difficult to devise an automatic procedure able to automatically determine which methods are worth of additional testing actions.

> In a sample of 101 pseudo-tested methods, systematically analyzed by the lead developers of 3 mature projects, 30 methods (30%) were considered worth of additional testing actions. The developer decisions are based on a deep understanding of the application domain and design of the application. This means that it is not reasonable to prescribe the absolute absence (zero) of pseudo-tested methods.

## 5 Threats to validity

**RQ1 and RQ2**. A threat to the quantitative analysis of RQ1 and RQ2 relates to external validity:

- Some extreme transformations could generate programs that are equivalent to the original. Given the nature of these transformations many of possible equivalent variants are detected by inspecting the method before applying the transformation. Methods with empty body and those returning a

constant value, are skipped from the analysis. This is a problem extreme transformations have in common with traditional mutation testing. The equivalent mutant problem could also affect the value of the mutation scores computed to answer RQ2.
- The values used to transform the body of non-void and non-boolean methods may affect their categorization as pseudo-tested or required. A different set of values may produce a different categorization. Since only one detected value is needed to label a method as required, then we actually produce an over-estimation of these methods. More values could be used to reduce the final set. In our study we find only 916 of non-void and non-boolean pseudo-tested methods which represents a 36% of the total number of methods. *void* and *boolean* methods tend to produce more pseudo-tested methods in our study subjects.
- As pointed by Goran and Ivankovic [20], mutation testing results can be affected by the programming language. This affects both, the extreme transformations performed and the mutation testing validation in RQ2. All our study subjects are Java projects, so our findings can not be generalized to other languages.
- We have considered all test cases equally and did not attempt to distinguish between unit and integration tests. We made this decision because such a distinction would require setting an arbitrary threshold above which a JUnit test case is considered an integration test. Yet, considering all test cases equally could influence the amount of pseudo-tested methods.

**RQ3 and RQ4**. The outcome of the qualitative analysis is influenced by our insight into each project, the insight of the developers consulted, the characteristics of each code base and the methods presented. Some of the teams showed more interest and gave more importance to the findings than others. This was expected. Not all developers had strong opinions regarding the presented issues.

## 6 Related work

Our work is inspired by the original paper on pseudo-tested methods [19]. The authors aim at assessing the relevance of test coverage with respect to test adequacy. They introduce the concept of pseudo-tested method to analyze the exact set of methods, which code is covered and which behavior is poorly assessed by the test suite. They also study the type of test involved with pseudo-tested methods, and aim at answering the question of whether code coverage is an indicator of test quality.

The novelty of our contribution with respect to this paper is three-fold. First, we perform a study with novel study objects (19 among the 21 projects studied here are not analyzed by Niedermayr and colleagues in this paper) and a different tool to detect the pseudo-tested methods. This mitigates both internal and external threats to the validity of Niedermayr's results. Second, we perform a novel study about the adequacy of the test suite for pseudo-tested

methods. Third, our most significant novel contribution consists in extensive exchanges and interactions with software developers to understand the type of testing issues that are revealed by pseudo-tested methods, as well as the characteristic of pseudo-tested that developers consider worth an additional testing effort.

As discussed in RQ3, pseudo-tested methods reveal weaknesses in the oracles of the test suite. Previous works have devised techniques to assess the quality of the test oracle. Schuler and Zeller [21] introduce the concept of checked coverage, as the percentage of program statements that are executed by the test suite and whose effects are also checked in the oracles. They compare this metric to code coverage and mutation testing with respect to their ability at assessing oracle decay. They perform manual checks on seven real software projects and conclude that checked coverage is more realistic than the usual coverage.

Jahangirova et al. [13] propose a technique for assessing and improving test oracles. They use mutation testing to increase the fault detection capabilities of the oracles and automated unit test generation to reduce the number correct executions rejected by the assertions. Their approach is shown to be effective in five real software projects. The fault detection ratio, approximated with the mutation score, is increased by 48.6% in average.

Staats et al. [23] extend the work of Gourlay [12] to provide a mathematical framework to capture how oracles interrelate with specifications, programs and tests. In this proposal an oracle is defined as a predicate whose domain is the product of the set of tests and the set of programs. Formal definitions of oracle completeness and soundness are given. The authors provide hints to the oracle selection problem and express criteria to compare oracles. The authors revisit concepts such as coverage and mutation testing under the light of their proposal.

The work of Androutsopoulos et al. [2] focuses on how faulty program states propagate to observable points in the program. They observe that one in ten test inputs fail to propagate to observable points. The authors provide an information theoretic formulation of the phenomenon through five metrics and experiment with 30 programs and more than 7M test cases. They state that better understanding the causes of failed error propagation leads to better testing.

Mutation testing is a well known technique to assess the fault detection capabilities of the test suite [7]. The traditional approach has been evaluated against real faults in several occasions [4, 1, 14]. The evidence presented supports that mutation testing is able to create effective program transformations under the assumption that programming errors are generally small and complex faults can be detected by tests which also detect simpler issues. However, some concerns has been raised regarding the conditions for those assumptions to be true [11].

Several works perform comparative analyses of mutation tools and confirm our choice of PITest for mutation analysis. Delahaye and Bousquet [5], then Kintis et al. [17] compare mutation tools from the usability point of view

concluding that PITest is one of the best alternatives for concurrent execution and adaptability to distinct requirements. In a follow-up paper, Laurent and colleagues [18] propose to improve PITest with an extended set of mutation operators shown to obtain better results.

More recently, Gopinath et al. [10] performed a comprehensive analysis of 3 software tools, including PITest, with 27 projects. They run several statistical analyses to compare the performance of the tools considering projects and tools characteristics against raw mutation score, a refined mutation score to mitigate the impact of equivalent mutants and the relationship among mutants. They conclude that PITest is slightly better than the other tools and that the specific project characteristics have a high impact on the effectiveness of the mutation analysis.

Other mutation tools, such as Major [15], could be extended as well to implement extreme transformations and detect pseudo-tested methods.

Several works from the mutation testing literature propose to transform programs using only deletion of statements, blocks, variables, operators and constants [6,8,24]. These approaches create much fewer mutants than traditional mutation operators showing decreases below 80%, but still remain at the instruction level which can difficult their understanding. Durelli et al. [9] actually state that these mutants are as time-consuming as the traditional operators when developers try to determine if they are equivalent to the original code. None of these works use extreme transformations as we have done here. Our work and our our exchanges with developers show that these transformations that delete the complete method body are easier to understand.

## 7 Conclusion

In this work, we provide an in-depth analysis of pseudo-tested methods found in open source projects. These methods, first coined by Niedermayr et al. [19], are intriguing from the perspective of the interaction between a program and its test suite: their code is executed when running the test suite, yet, none of their effects are assessed by the test cases.

Our key findings are as intriguing as the original concept. First, we observe that all 21 mature Java projects that we study include such methods: from 1% to 46% in our dataset. Second, we confirm that pseudo-tested methods are poorly tested, compared to the required methods: the mutation score of the former is systematically and significantly lower than the score of the latter. Third, our in-depth qualitative analysis of pseudo-tested methods and feedback from developers reveals the following facts:

- We assessed the relevance of pseudo-tested methods as concrete hints to reveal weak test oracles. These issues in the suite were confirmed by the developers, who accepted the pull requests that we proposed, to fix weak oracles.
- Less than 30% of pseudo-tested methods in a sample of 101 represent an actual hint for further actions to improve the test suite. Among the 70%

considered as worthless an additional testing action we found methods not widely used in the code base, automatically generated code, trivial methods, helper methods for debugging and receiving methods in delegation patterns.
- The pseudo-tested methods that actually reveal an issue in the interaction between the program and its test suite are involved in core functionalities, are widely used in the code base, are used by external clients or verify preconditions on input data.

In the light of these conclusions, the immediate next step in our research agenda is to investigate an automatic test generation technique targeted towards pseudo-tested methods. This technique shall kill two birds with one stone: improve the adequacy of the test suite for pseudo-tested methods; let the developers focus their efforts on core features and relieve them from the test improvement task. The current state of PITest and Descartes allows to exclude methods from the analysis based on the name, on some structural patterns, or considering the annotations marking the method. These filters are not able to express some of the criteria exposed in Section 4.4, for example, methods which are not widely used in the codebase. A future line will consist in improving the static analysis capacities of the tools to reduce even more the number of methods that developers do not consider worhty for testing. Further steps in the characterization of pseudo-tested methods could include the comparison of the statement coverage ratio between pseudo-tested and required methods, in the same way we have done considering the method-level mutation score. Another future task could be to assess the fault detection capabilities of extreme transformations in the same way it has been done for mutation operators [11].

## Acknowledgement

We would like to acknowledge the invaluable help and feedback provided by the development teams of `authzforce`, `spoon` and `sat4j-core`. We also express our appreciation to Simon Urli, Daniel Le Berre, Arnaud Blouin, Marko Ivanković, Goran Petrovic and Andy Zaidman for their feedback and their very accurate suggestions. This work has been partially supported by the EU Project STAMP ICT-16-10 No.731529 and by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

## References

1. J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.
2. K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, and M. Harman. An analysis of the relationship between conditional entropy and failed error propagation in software

testing. In *Proceedings of the 36th international conference on software engineering*, pages 573–583. ACM, 2014.
3. H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452, New York, NY, USA, 2016. ACM.
4. M. Daran and P. Thévenod-Fosse. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 158–171, New York, NY, USA, 1996. ACM.
5. M. Delahaye and L. d. Bousquet. A Comparison of Mutation Analysis Tools for Java. In *2013 13th International Conference on Quality Software*, pages 187–195, July 2013.
6. M. E. Delamaro, J. Offutt, and P. Ammann. Designing deletion mutation operators. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 11–20, March 2014.
7. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
8. L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 84–93, March 2013.
9. V. H. S. Durelli, N. M. D. Souza, and M. E. Delamaro. Are deletion mutants easier to identify manually? In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 149–158, March 2017.
10. R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce. Does choice of mutation tool matter? *Software Quality Journal*, 25(3):871–920, Sept. 2017.
11. R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Software reliability engineering (ISSRE), 2014 IEEE 25th international symposium on*, pages 189–200. IEEE, 2014.
12. J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on software engineering*, (6):686–709, 1983.
13. G. Jahangirova, D. Clark, M. Harman, and P. Tonella. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 247–258. ACM, 2016.
14. R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are Mutants a Valid Substitute for Real Faults in Software Testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.
15. R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 612–615, November 9–11 2011.
16. V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information & Software Technology*, 49(11-12):1073–1086, 2007.
17. M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris. Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–156, Oct. 2016.
18. T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque. Assessing and Improving the Mutation Testing Practice of PIT. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435, Mar. 2017.
19. R. Niedermayr, E. Juergens, and S. Wagner. Will my tests tell me if I break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pages 23–29, New York, NY, USA, 2016. ACM Press.
20. G. Petrovic and M. Ivankovic. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering 2017 (SEIP)*, 2018.
21. D. Schuler and A. Zeller. Checked coverage: an indicator for oracle quality. *Software Testing, Verification and Reliability*, 23(7):531–551, 2013.

22. F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo. The role of replications in Empirical Software Engineering. *Empirical Software Engineering*, 13(2):211–218, Apr. 2008.
23. M. Staats, M. W. Whalen, and M. P. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd international conference on software engineering*, pages 391–400. ACM, 2011.
24. R. H. Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM.
25. O. L. Vera-Pérez, M. Monperrus, and B. Baudry. Descartes: A pitest engine to detect pseudo-tested methods. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, pages 908–911, 2018.

## A Source code for the study subjects

**Legend.** Column "Project" list the projects included in the present study. Column "URL" contains links to the available source code. Column "Commit ID" contains the SHA-1 hash identifying the commit with the source code state that was used in this study.

| Project | URL/Commit ID |
|---|---|
| authzforce | https://github.com/authzforce/core.git<br>81ae56671bc343eabf2bc99ee0c51ba6ae28d649 |
| aws-sdk-java | https://github.com/aws/aws-sdk-java<br>b5ae6ce44f4b5053a9a0255c9648f3073fafcf55 |
| commons-cli | https://github.com/apache/commons-cli<br>c246bd419ee0efccd9a96f9d33486617d5d38a56 |
| commons-codec | https://github.com/apache/commons-codec<br>e9da3d16ae67f2940a0bbdf982ecec19a0481981 |
| commons-collections | https://github.com/apache/commons-collections<br>db189926f7415b9866e76cd8123e40c09c1cc67e |
| commons-io | https://github.com/apache/commons-io<br>e36d53170875d26d59ca94bd376bf40bc5690ee6 |
| commons-lang | https://github.com/apache/commons-lang<br>e8f924f51be5bc8bcd583ea96e5ef25f9b2ca72a |
| flink-core | https://github.com/apache/flink/tree/master/flink-core<br>740f711c4ec9c4b7cdefd01c9f64857c345a68a1 |
| gson | https://github.com/google/gson<br>c3d17e39f1cb6ec41496e639ab42f7e7cca3b465 |
| jaxen | https://github.com/jaxen-xpath/jaxen<br>a8bd80599fd4d1c9aa1248d3276198535a30bfc5 |
| jfreechart | https://github.com/jfree/jfreechart<br>a7156d4595ff7f6a7c8dac50625295c284b86732 |
| jgit | https://github.com/eclipse/jgit<br>1513a5632dcaf8c6e2d6998427087e11ba35566d |
| joda-time | https://github.com/JodaOrg/joda-time<br>6ad133837a4c4f8199d00a05c3c16267dbf6deb8 |
| jopt-simple | https://github.com/jopt-simple/jopt-simple<br>b38b70d1e7685766ab400d8b57ef9ca9c010e0bb |
| jsoup | https://github.com/jhy/jsoup<br>35e80a779b7908ddcd41a6a7df5f21b30bf999d2 |
| pdfbox | https://github.com/apache/pdfbox<br>09e9173e400ea6c1794354ca54c36607c53391ff |
| sat4j-core | https://gitlab.ow2.org/sat4j/sat4j/tree/master/org.sat4j.core<br>1a0127645bf98b768ee3628076d0246596dd15eb |
| scifio | https://github.com/scifio/scifio<br>2760af6982ad18aab400e9cd99b9f63ef2495333 |
| spoon | https://github.com/INRIA/spoon<br>fd878bc71b73fc1da82356eaa6578f760c70f0de |
| urbanairship | https://github.com/urbanairship/java-library<br>aafc049cc1cd3971c62a3dfc1d72cfe61160f32c |
| xwiki-rendering | https://github.com/xwiki/xwiki-rendering<br>cb3c444fb743e073eefbac2b44351a6166d94ac1 |