

Will My Tests Tell Me If I Break This Code?*

Rainer Niedermayr, Elmar Juergens
CQSE GmbH
Garching b. München, Germany
{niedermayr, juergens}@cqse.eu

Stefan Wagner
University of Stuttgart
Stuttgart, Germany
stefan.wagner@informatik.uni-
stuttgart.de

ABSTRACT

Automated tests play an important role in software evolution because they can rapidly detect faults introduced during changes. In practice, code-coverage metrics are often used as criteria to evaluate the effectiveness of test suites with focus on regression faults. However, code coverage only expresses which portion of a system has been executed by tests, but not how effective the tests actually are in detecting regression faults.

Our goal was to evaluate the validity of code coverage as a measure for test effectiveness. To do so, we conducted an empirical study in which we applied an extreme mutation testing approach to analyze the tests of open-source projects written in Java. We assessed the ratio of pseudo-tested methods (those tested in a way such that faults would not be detected) to all covered methods and judged their impact on the software project. The results show that the ratio of pseudo-tested methods is acceptable for *unit* tests but not for *system* tests (that execute large portions of the whole system). Therefore, we conclude that the coverage metric is only a valid effectiveness indicator for unit tests.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Regression Testing, Test Suite Effectiveness, Code Coverage, Mutation Testing.

1. INTRODUCTION

Code might get unintentionally broken during software evolution. Automated tests ensure the correctness of the

*This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant “Q-Effekt, 01IS15003A”. The responsibility for this article lies with the authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CSED '16 May 14–15, 2016, Austin, TX, USA

© 2016 ACM. ISBN 000-0000-00-000/00/00.

DOI: 00.000/000_0

software under test and can reveal newly introduced faults in code chunks that previously worked properly. Automated tests are executed regularly (periodically or after each commit) in a development process with continuous integration. Thereby, faults can be discovered at an early stage in the software life cycle, saving failure follow-up costs.

An important question is how good are these automated tests at revealing regression faults. A measure of how likely test cases will detect broken code helps us better allocate quality-assurance (QA) efforts. Code coverage metrics are often used in practice for judging test effectiveness.

Problem Statement. Code coverage only measures which portion of the whole code was *executed* by tests. It does not take into account if a method was *tested* with appropriate assertions or if it was executed without any assertions or incomplete ones. It is unclear if code coverage is a valid effectiveness indicator, or if it is misleading and provides a false sense of effectiveness.

Research Objective. Continuous integration uses automated tests to detect regression faults. Therefore, we need a good understanding of the effectiveness for the fault-detection capability of test cases. Our goal is to support continuous integration by evaluating if code coverage at the method level is a valid measure for the test effectiveness of unit and system tests.

We conducted an experiment in which we analyzed 14 open-source projects. First, we used an extreme mutation testing approach to remove the logic of the methods and re-run the test cases. We then assessed the number of methods with which the test cases still run successfully and investigated the role these methods play in their respective system.

Contribution. In this paper, we show that code coverage at the method level is a valid indicator for the effectiveness of unit tests but not for system tests.

This code snippet demonstrates that high code coverage does not imply test effectiveness:

```
public class Calculation {
    private int value;

    public Calculation() {
        this.value = 0;
    }
    public void add(int x) {
```

```

    this.value += x;
}
public boolean isEven() {
    return this.value % 2 == 0;
}
}

```

The class `Calculation` consists of an integer field named `value` and two public methods. The `add` method allows for adding an integer to the internal value. The `isEven` method returns a boolean value, which indicates if the current internal value is an odd or even number. This class is tested by a JUnit test in the `CalculationTest` class.

```

public class CalculationTest {
    @Test
    public void testCalculation() {
        Calculation calc = new Calculation();
        calc.add(6);
        assertTrue(calc.isEven());
    }
}

```

The test case creates a new instance of the `Calculation` class and implicitly assigns 0 to the field `value`. It then uses the `add` method to increase the internal value by 6. Finally, the test case verifies that the `isEven` method returns true, which is expected for 6.

The test case executes all methods, statements, and branches of the class under test. This results in 100% code coverage at the method, statement and branch levels. Consequently, one could assume that the `Calculation` class is effectively tested.

However, this is not the case. Let's assume that the programmer forgot to implement the logic of the `add` method so that its body is empty. Although the test case covers that method, it will not detect the fault, because both 0 and 6 are even numbers. Therefore, the `add` method is executed, but not effectively tested, because the test case would not detect any faults. We call this a pseudo-tested method.

According to Fowler [4], test cases that do not contain any assertions are another example of tests that increase the coverage, but are useless (unless their purpose is to check if exceptions are thrown).

2. FUNDAMENTALS AND TERMS

This section describes the terms used in this paper.

A *unit test* examines a small unit of code (usually a method or a class). It consists of a sequence of method calls and assertions that verify that the computed results of the invocations equal the expected ones. A unit test can also check the absence of thrown exceptions for a given program flow.

A *system test* examines a complete software system, which may consist of several components. Unlike a unit test, it covers many methods by executing a large proportion of the whole system. A system test often triggers the execution of a large functionality and compares the end result (which can be aggregated data, a report, or a log file) with the expected one.

Code coverage is a metric that expresses which ratio of application code of a software project is executed when run-

ning all test cases. It can be computed at different levels; widely used are measures at method/function, statement or branch level. When we refer to code coverage in this paper, we mean method coverage. It corresponds to the ratio of methods that are executed by tests.

We consider a method to be *test-executed* if it is covered by at least one test case. A test-executed method is considered *tested* if at least one covering test case fails when the whole logic of the method is removed. In contrast, a test-executed method is considered *pseudo-tested* if none of the covering test cases fails when the whole logic of the method is removed.

We define *test effectiveness* as the capability of test cases to detect regression faults in methods that they execute. Intuitively, we understand the overall ratio of code that is effectively tested as the upper bound of the probability that test cases detect a novel regression fault in a project.

3. RELATED WORK

This paper is based on the master's thesis of the first author [11]. Related work is in the areas of code coverage, test suite effectiveness and mutation testing.

3.1 Code Coverage, Test Suite Effectiveness

Wong, Horgan, London and Mathur [12] showed that the correlation between fault detection effectiveness and block coverage is higher than between effectiveness and the size of the test set. This indicates that coverage can be a valid measure for test effectiveness. However, they did not differentiate between different test types.

In [2], Andrew, Briand, Labiche and Namin conducted an empirical study on one industrial program with known faults to investigate test coverage criteria on fault-detection effectiveness. Their results showed that no one coverage criteria is more cost-effective than any other, but more demanding criteria lead to larger test suites that detect more faults. In contrast, we analyzed open-source systems and made use of mutation testing.

In [9], Mockus, Nagappan and Dinh-Trong revealed that an increase in coverage exponentially increases the test effort and linearly reduces the field problems. They suggested that "code coverage is a sensible and practical measure of test effectiveness", but did not differentiate between unit and system tests.

Namin and Andrews [10] studied the relationship between size, coverage, and fault-finding effectiveness of test suites. They found that both size and coverage are important for effectiveness and suggested a nonlinear relationship between them. They analyzed very small C programs (the largest one consisted of less than 6,000 lines of code).

Inozemtseva and Holmes [7] came to different conclusions. They evaluated the relationship between test suite size, coverage, and effectiveness for Java programs. They found that: "High levels of coverage do not indicate that a test suite is effective." In addition, they discovered that the type of coverage had little effect on the correlation between coverage and effectiveness. They also used mutation testing, but they generated test suites of a fixed size by randomly selecting test cases.

Marick [8] critically analyzed code coverage as a metric for test effectiveness. He showed how code coverage is com-

monly misused and argued for a cautious approach to the use of coverage. His work did not assess the validity of code coverage as effectiveness indicator.

3.2 Mutation Testing

Mutation testing was first proposed in the 1970s and is an established, powerful technique to evaluate test suites. The general principle is to generate mutants by introducing faults into a program and check if the tests can detect (*kill*) these. Experimental studies ([1], [3], [6], [?]) provide evidence that mutation testing is a good indicator for the fault-detection capability of test suites.

Mutation testing has two major drawbacks, which explain why it is not widely used in practice. First, the computational costs are high because mutation testing involves creating a large number of mutants and the execution of all tests that cover the mutated code chunk for each mutant. Second, some of the generated mutants are semantically equivalent to the original code. The so-called *equivalent mutants* do not represent injected faults, cannot be killed by tests and so distort the results. The detection of these mutants is generally undecidable. Grün, Schuler and Zeller [5] identified equivalent mutants as an important problem because they are surprisingly common. Our approach addressed these issues and is explained in the next section.

4. MUTATION APPROACH

The general idea is to apply mutation testing as mean to collect data for assessing test effectiveness. We used an extreme mutation operator in which we eliminated the whole logic from a method and determined whether or not a method is pseudo-tested. Therefore, we got around the two drawbacks of mutation testing (mentioned in Section 3.2). First, we created, at most, two different mutants for a method, keeping the number of mutants manageable and the execution time for analyzing a medium-sized software project within a few hours. Second, the mutation operator radically changes the methods and generates less equivalent mutants. The majority of the generated equivalent mutants can be identified automatically.

Our approach consists of four steps and is depicted in Figure 1. The first two steps are executed once and are necessary to collect information about the test cases. The latter two steps comprise the actual mutation process. They are executed for each method under test and can be run concurrently.

In *Step 1*, the project code is instrumented. This is done by inserting logging statements.

In *Step 2*, all test cases are executed once on the instrumented code. This allows us to determine the relationships between test cases and methods. From the information of the test-executed methods for a given test case, the opposite direction of the relation (all test cases covering a given method) can be computed. Test cases that fail at this point are excluded from further analysis.

In *Step 3*, the mutation of one method under test is performed. The mutation operator removes the whole logic of the method. This is done as follows:

- For void methods, all statements are removed. No further actions are necessary.

Table 1: Return values for primitive types and string

| Return Type | Mutant 1 | Mutant 2 |
|------------------------|----------|----------|
| boolean | false | true |
| byte, short, int, long | 0 | 1 |
| float, double | 0.0 | 1.0 |
| char | ' ' | 'A' |
| string | "" | "A" |

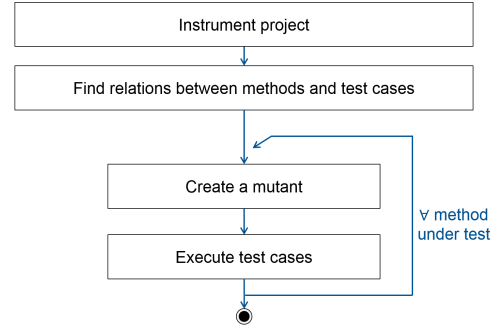


Figure 1: Steps of the mutation analysis process. The first two steps are executed once, the latter ones are executed for each considered method.

- For methods that return a primitive or a string value, two mutants are generated. The mutation removes the original code by replacing it with a return statement. The value to be returned depends on the return type and is different for both created mutants. Table 1 lists the return values for the primitive types and string.
- For methods that return an object, a factory is used to generate a suitable instance. The original code is removed and the generated instance is returned. We developed factories for three study objects (see Section 5.4).

A method is excluded from the mutation analysis if it:

- does not contain any statements (otherwise an equivalent mutant would be generated).
- is a setter or getter method consisting of exactly one statement (we consider such a method as too trivial to test).
- is a constructor.
- is a special method generated by the compiler (such as Java synthetic methods).
- returns an object but no factory is provided, or the factory cannot create an appropriate instance.

In *Step 4*, all test cases that cover the mutated method are executed against the mutated code. The outcome of this step shows which test cases fail and which still succeed after the mutation.

5. EXPERIMENTAL DESIGN

We analyze open-source projects to investigate pseudo-testing and the validity of the code-coverage metric at the method level.

5.1 Study Objects

To perform the experiment, we chose 14 open-source projects. The projects satisfied the following criteria: They

Table 2: Study Objects

| Name | LOC | Tests | Test Type |
|----------------------------|---------|----------------|-----------|
| Apache Commons Collections | 109,415 | 4,468 | unit |
| Apache Commons Lang | 100,422 | 2,000 | unit |
| Apache Commons Math | 275,570 | 3,463 | unit |
| Apache Commons Net | 53,601 | 133 | unit |
| ConQAT Engine Core | 27,481 | 107 | unit |
| ConQAT Lib Commons | 43,419 | 468 | unit |
| ConQAT dotnet | 8,239 | 22 | system |
| DaisyDiff | 11,288 | 1 ¹ | system |
| Histone | 24,412 | 93 | system |
| LittleProxy | 7,300 | 93 | system |
| Predictor | 7,733 | 21 | system |
| Struts 2 | 148,486 | 6 | system |
| Symja | 443,092 | 448 | system |
| Tspmcab | 44,999 | 10 | system |

must use Java as a programming language because the mutation approach is performed on the Java bytecode. Moreover, the projects must contain tests that use either the JUnit or testNG framework.

The selected projects are of different sizes. The smallest one consists of about 7,000 lines of code (LOC), and the largest project measures about 500,000 LOC. The projects can be classified into libraries with unit tests and systems with system tests. For systems, we considered only system tests, even if they also had unit tests. Table 2 lists the projects and their characteristics.

5.2 Research Questions

In this experiment, we wanted to determine the validity of code coverage as a criterion for test effectiveness. The effectiveness expresses how likely it is that newly introduced faults in test-executed methods are revealed by test cases.

RQ 1: What is the ratio of pseudo-tested methods?

Research Question 1 evaluates how many methods are test-executed but are actually only pseudo-tested. As pseudo-tested methods contribute to the overall code coverage of a software project, the amount of test-executed code suggested by the code-coverage metric is higher than the amount of (effectively) tested code. Therefore, code coverage might not be a valid indicator for test effectiveness.

RQ 2: Does the ratio of pseudo-tested methods depend on the type of test?

Research Question 2 investigates the influence of the type of test (unit tests versus system tests) on the ratio of pseudo-tested methods. We expect methods that are test-executed by system tests to more likely be pseudo-tested because system tests execute many methods in one run. Therefore, we want to find out if code-coverage validity should be assessed separately for these two types of tests.

RQ 3: How severe are the pseudo-tested methods?

Research Question 3 analyzes the functional purpose and the severity of pseudo-tested methods. We want to understand how severe is the lack of test effectiveness of these methods for a project. Some methods may not warrant being tested (because they are too trivial or non-deterministic),

¹The single system test is parameterized and executed with 247 different input files.

but others may be relevant and need more thorough testing to detect regression faults.

5.3 Experiment Design

First, we ran the mutation testing analysis for the study objects using our Java program. It already comprises the exclusion of provably equivalent mutants and simple setters and getters (see Section 4). Then, we analyzed the obtained data to answer the research questions. The design for each research question is as follows:

RQ 1. We denoted the number of pseudo-tested methods as M_{PT} and calculated the ratio of pseudo-tested methods as follows:

$$r(M_{PT}) = \frac{\text{number of pseudo-tested methods } (M_{PT})}{\text{number of mutated test-executed methods}}$$

The mutated test-executed methods comprise all methods for which a mutant is created and tested (by at least one test case without timeout) during the mutation analysis (refer to Section 4 for excluded methods).

Consequently, the ratio of tested methods is:

$$r(M_T) = 1 - r(M_{PT})$$

We also presented the method coverage (CC) of the study objects and computed the overall ratio of tested methods to all existing methods:

$$r(CT) = CC * r(M_T)$$

(assuming that the test-executed methods that were excluded from the mutation analysis provide an approximately similar result). We considered $r(CT)$ as an upper bound for the ratio of tested code to the whole project. Test cases can find faults in this portion of the project.

RQ 2. To answer this question, we examined the ratio of pseudo-tested methods for unit and system tests separately. We then compared the mean ratio of pseudo-tested methods per test type, computed the standard deviation, and compared the data using boxplots.

RQ 3. We investigated all pseudo-tested methods and assigned each one to a functional category. The mapping was done by manual inspection based on the method name. We then assigned a severity to each functional category. Table 3 presents the categories and their severities. The severities are defined as follows:

- Functional categories with methods that are not deterministic (such as generating a random number) or not intended to be tested were assigned a severity of *irrelevant*. Moreover, unless the hash generation is explicitly tested, the `hashCode()` method belonged to the severity *irrelevant* because the logic of this method still corresponds with its specification after the mutation if it always returns the same constant value.
- The severity *low* is assigned to functional categories that contain methods that do not significantly influence the program execution. The most prominent examples are validation and optimization methods, as well as those to close streams and connections.
- The severity *medium* was for categories with methods that are likely to influence the program execution to

Table 3: Functional categories and their severity

| Functional Category | Examples |
|---------------------|---|
| Irrelevant | |
| hashcode | <code>hashCode()</code> |
| non-deterministic | <code>setSeed(int)</code> |
| test-related | <code>updateTestData()</code> |
| Low severity | |
| finalization | <code>finalize()</code> , <code>closeStream()</code> |
| monitoring | <code>logInfo(String)</code> |
| optimization | <code>estimateLength()</code> , <code>addToCache(Object)</code> |
| validation | <code>checkIndex(int)</code> , <code>validateParam(Object)</code> |
| Medium severity | |
| events | <code>notifyListeners()</code> , <code>firePropertyChange()</code> |
| preparation | <code>initWorkflow()</code> , <code>setUpBlock()</code> |
| setter and getter | <code>isRed(Color)</code> , <code>getV(int)</code> |
| toString | <code>toString()</code> |
| transformation | <code>abs(int)</code> , <code>escape(String)</code> |
| High severity | |
| object identity | <code>equals(Object)</code> , <code>compareTo(T)</code> |
| program logic | <code>computeLSB()</code> , <code>solvePhase1()</code> |

some extent. These include methods that send events to listeners, prepare a computation, or set or get properties², or transform or convert a value. We also assigned the `toString()` method that returns a string representation of an object to a severity of *medium*.

- The severity *high* was assigned to categories that contain methods likely to be very relevant for the program execution. They concern the computation logic or important data structures. Examples are: `calculateAST()`, `writeToFile()`, `storeObject()`.

Moreover, this severity comprises the `equals(Object)` method, which checks if an object is semantically equal to another one, and the `compareTo(T)` method, which returns the natural order of two objects.

5.4 Experiment Procedure

This section describes the mutation testing analysis execution for the study objects.

We checked out the source code of the projects from the repositories; built the code with Ant, Maven, or Gradle according to the instructions; and imported the projects in the Eclipse IDE. We then looked at the source code to select the unit or system tests. We exported the compiled application and test code as separate jar files and provided the necessary test data. We specified the location of the jar files and other dependencies in a configuration file. Moreover, we defined timeouts for the test cases (twice as long as the longest duration of any test case on the original code) and the number

²Note that our analysis results do not contain very simple setters and getters consisting of a single statement (as described in Section 4).

Table 4: Overview of results

| Study Object | M_{PT} | $r(M_{PT})$ | CC | $r(CT)$ |
|--------------------|----------|-------------|-------|---------|
| Apache Comm. Coll. | 124 | 9.5% | 81.6% | 73.9% |
| Apache Comm. Lang | 22 | 1.9% | 93.0% | 91.3% |
| Apache Comm. Math | 271 | 10.6% | 84.8% | 75.8% |
| Apache Comm. Net | 28 | 18.4% | 29.0% | 23.7% |
| ConQAT Engine Core | 41 | 18.9% | 50.0% | 40.5% |
| ConQAT Lib Commons | 45 | 9.5% | 56.3% | 51.1% |
| ConQAT dotnet | 154 | 36.3% | 48.1% | 30.7% |
| Daisydiff | 7 | 6.4% | 49.8% | 46.7% |
| Histone | 47 | 24.8% | 73.0% | 54.9% |
| LittleProxy | 35 | 71.4% | 45.4% | 13.0% |
| Mut. Testing Prot. | 7 | 25.0% | 73.1% | 54.8% |
| Predictor | 80 | 52.7% | 72.4% | 34.2% |
| Struts 2 | 154 | 45.9% | 27.0% | 14.6% |
| Symja | 234 | 25.0% | 21.3% | 15.9% |
| Tspmcabe | 13 | 21.3% | 39.1% | 30.7% |

of concurrently running tests. Some study objects required the development of a tailored test runner to locate the test cases and run them piecewise.

We then executed the mutation testing analysis for methods with void, primitive or string as return types. We analyzed the method signatures of the study objects and discovered that about 50% to 65% of the methods belonged to this group. We also developed factories to create instances for three study objects (*Apache Commons Lang*, *Apache Commons Collections*, and *ConQAT Engine Core*) to investigate methods that return objects in a separate analysis. Since the gained results were approximately comparable to the results of methods with a primitive return type, we did not further investigate methods returning objects in this experiment.

After the completion of the mutation analysis, we looked at the log file to check for any unexpected problems. If serious problems were logged, we fixed the cause and restarted the analysis. Finally, we imported the analysis results in the form of SQL statements into a database and stored additional execution information (such as the duration).

6. RESULTS

The answers to the research questions suggest that code coverage is a valid effectiveness indicator for unit tests. This does not apply to system tests because the ratio of pseudo-tested methods is higher for this type of test and heavily deviates, depending on the project.

RQ 1: What is the ratio of pseudo-tested methods?

The ratio of pseudo-tested methods varied heavily and ranged between 6% and 53% for most study objects. According to the results, the *Apache Commons Lang* methods are mostly effectively tested (according to the definition in Section 2) because the code coverage at the method level was 93% and less than 2% of the test-executed methods were classified as pseudo-tested. Other study objects exhibited a much higher ratio of pseudo-tested methods, including the *Predictor*, *Struts 2*, and *LittleProxy* projects with more than half of the test-executed methods being pseudo-tested.

Table 4 presents the number M_{PT} and ratio $r(M_{PT})$ of pseudo-tested methods, the method coverage CC , and the overall ratio of tested methods $r(CT)$ for each study object.

RQ 2: Does the ratio of pseudo-tested methods de-

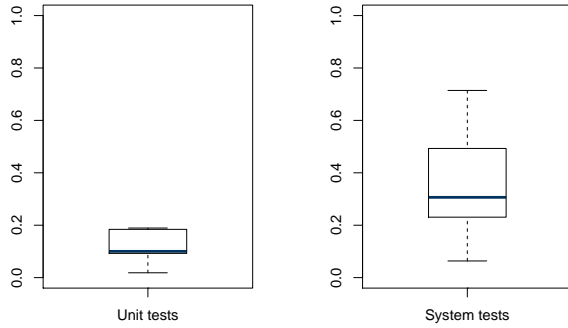


Figure 2: Boxplot comparing the ratios of pseudo-tested methods between unit and system tests

pend on the type of test?

The mean ratio of pseudo-tested methods of the study objects differs for unit and system tests. The mean ratio was 11.41% for unit tests and 35.48% for system tests. It is striking that the ratio is within a small range for unit tests but greatly deviates among systems tests. The standard deviation of the ratio of pseudo-tested methods (6.42% for unit and 20.60% for system tests) confirms this observation. The boxplot in Figure 2 depicts the key figures and the deviation between unit and system tests. The results show that the type of test influences the ratio of pseudo-tested methods.

RQ 3: How severe are the pseudo-tested methods?

Figure 3 presents the absolute and relative number of pseudo-tested methods, grouped by severity. For 11 study objects, more than half of the pseudo-tested methods were of medium or high severity. This was not the case for *Apache Commons Math*, which contains a significant amount of irrelevant pseudo-tested methods (some test utility methods were mutated in the analysis); and *Apache Commons Lang* as well as *ConQAT Lib Commons* with some low-importance methods. The results confirm the relevance of pseudo-tested methods and suggest that the lack of test effectiveness is a problem for a software project.

7. THREATS TO VALIDITY

We separated the threats to validity into internal and external threats.

7.1 Internal Threats

The threats to internal validity comprise reasons why the results could be invalid for the study objects.

One threat to the internal validity is that some methods considered pseudo-tested might actually result in equivalent mutants. We tried to mitigate this issue by the choice of the mutation operator and additional filtering. The mutation operator modifies the whole method body, while many common operators only mutate single lines, and is therefore less likely to create an equivalent mutant. Additionally, we filtered out empty and trivial methods such as one-line setters and getters (see Section 4). As we generated two mutants for methods with primitive or string return types, the results were not distorted if only one mutant was equivalent. We manually reviewed a random sample to make sure that the number of remaining undetected equivalent mutants was negligible.

Study objects with test cases that fail on the original code are a further threat to internal validity. This can happen because of the test environment or faulty code in the study object. Some test cases rely on further data stored in files, a database with a certain data model and content, other connected systems, or the network connection. We tried to supply all the needed and available files in the test execution folder and set up the databases according to the project manuals. Nevertheless, some test cases still failed. This was the case for the *Apache Commons Net* project, which presumably required certain firewall settings for some of its test cases. We excluded these failing test cases from the analysis. If the excluded test cases had worked, they might have killed some mutants that were not killed by the other test cases (and therefore categorized as pseudo-tested). For this reason, we only selected projects as study objects in which most test cases could be successfully executed on the original code.

Another threat to internal validity is custom class loaders that could interfere with the test execution on mutated code and affect the results in some seldom cases. This may occur if a class is loaded multiple times by different class loaders during the execution of a single test case. In this case, another class version is loaded in addition to the mutated one. We consider this threat to be negligible.

One threat regarding the definition of test effectiveness is the fact that test cases can reveal faults that cause an exception to be thrown, even in pseudo-tested methods (that are considered as not effectively tested).

Concerning Research Question 3, the categorization of pseudo-tested methods according to their purpose was performed by considering the method name. Due to the high number of methods, it was not feasible to inspect all the code to determine their purpose. Therefore, some methods might actually belong to another category and severity than the assigned one.

7.2 External Threats

The external validity concerns the generalization of the results of the experiment. One threat to external validity is that analyses were performed only for void methods and methods returning a primitive or string value. Although we additionally executed the analysis for three study objects with methods returning objects and observed comparable results, the obtained results might not be representative of methods returning objects.

Furthermore, the results of the selected open-source projects might not be representative of closed-source systems. We tried to mitigate this issue by considering several projects with different characteristics and application domains as study objects. Further studies are necessary to determine if the results also apply to closed-source systems.

8. CONCLUSIONS AND FUTURE WORK

The results of the experiment show that approximately 9% to 19% of the test-executed methods are pseudo-tested in projects with unit tests. The ratio does not heavily deviate among study objects with unit tests (mean: 11.41%, standard deviation: 6.42%). Therefore, code coverage at the method level is not completely misleading and can be used as an approximation for the effectiveness of unit tests.

In contrast, the ratio of pseudo-tested methods for system tests is generally higher than the ratio for unit tests. It

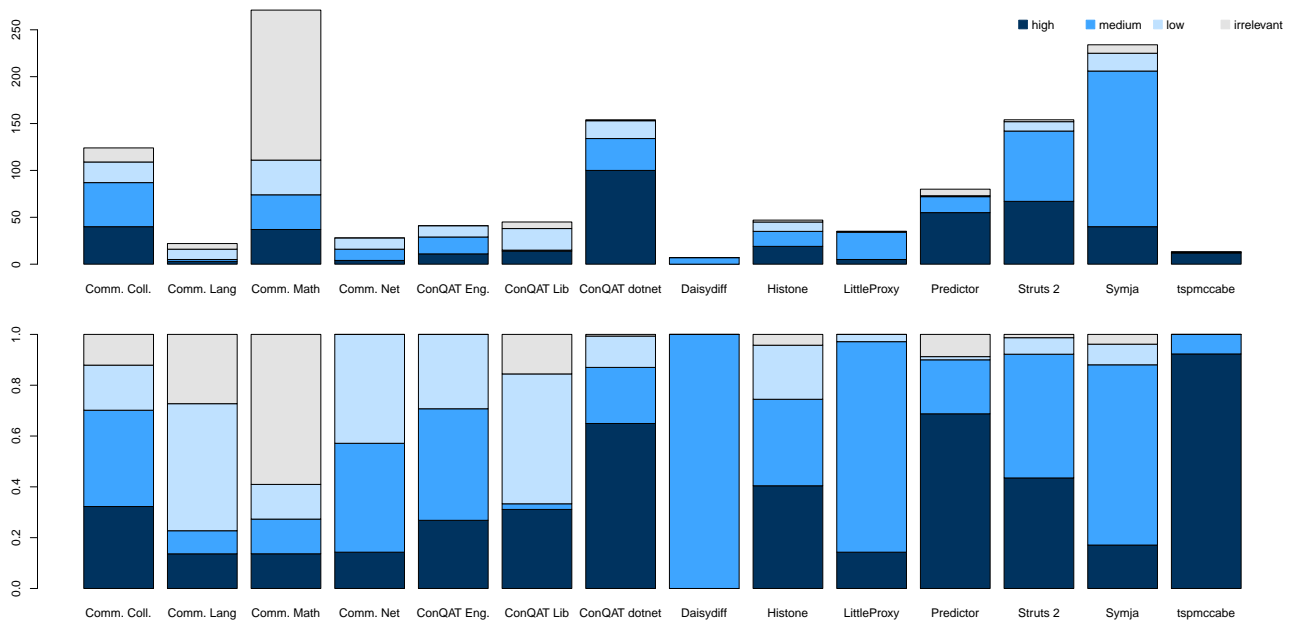


Figure 3: Pseudo-tested methods by their severity (top: absolute numbers, bottom: ratios)

ranges between 6% and 72% for the analyzed study objects and heavily deviates (mean: 35.48%, standard deviation: 20.60%). Therefore, code coverage at the method level is not a valid indicator for the effectiveness of system tests.

The assessment of the functional purpose and severity of pseudo-tested methods confirms the relevance of these methods for the software. Faults in these methods would not be detected and could cause failures.

For future work, we intend to investigate characteristics of pseudo-tested methods and their relationship to test cases. We want to find indicators that reveal pseudo-tested methods. Such indicators would enable rapid detection of these methods in a static code analysis and make the computationally expensive mutation testing approach superfluous. This static code analysis could support developers in a continuously integrated development process. Moreover, we plan to replicate the experiment with closed-source study objects.

9. REFERENCES

- [1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. 27th International Conference on Software Engineering (ICSE)*. IEEE, 2005.
- [2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8), 2006.
- [3] M. Daran and P. Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *ACM SIGSOFT Software Engineering Notes*, volume 21. ACM, 1996.
- [4] M. Fowler. AssertionFreeTesting. <http://martinfowler.com/bliki/AssertionFreeTesting.html>. Visited on December 21st, 2015.
- [5] B. J. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Proc. International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2009.
- [6] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proc. 16th International Conference on Software Engineering (ICSE)*. IEEE, 1994.
- [7] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. 36th International Conference on Software Engineering (ICSE)*. ACM, 2014.
- [8] B. Marick. How to misuse code coverage. <http://www.exampler.com/testing-com/writings/coverage.pdf>. Visited on January 14th, 2016.
- [9] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proc. 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2009.
- [10] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proc. 18th International Symposium on Software Testing and Analysis*. ACM, 2009.
- [11] R. Niedermayr. Meaningful and practical measures for regression test reliability. Master’s thesis, Technische Universität München, Munich, Germany, 2013.
- [12] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software: Practice and Experience*, 26(2), 1996.
- [13] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Proc. 5th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 1994.