



Universität Stuttgart

# Evaluation and Improvement of Automated Software Test Suites

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik  
der Universität Stuttgart zur Erlangung der Würde eines Doktors der  
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von  
**Rainer Niedermayr**  
aus Bozen, Italien

**Hauptberichter:** Prof. Dr. Stefan Wagner

**Mitberichter:** Prof. Dr. Benoit Baudry

**Tag der mündlichen Prüfung:** 7. Oktober 2019

Institut für Softwaretechnologie

2019



# ACKNOWLEDGMENTS

This dissertation was written under the supervision of Prof. Dr. Stefan Wagner between 2015 and 2019. I am very grateful for his mentorship, academic guidance, and advice. In addition, I am thankful for my inclusion in the research group at the Institute of Software Technology at University of Stuttgart and want to thank all of the members for their support and the discussions at the seminars.

My sincere thanks also go to Prof. Dr. Benoit Baudry for co-supervising this dissertation, inspiring discussions in Gothenburg and Stockholm, and his valuable feedback on my work.

I am grateful for the support of Dr. Elmar Juergens, who had already been advisor of my Master's thesis, for encouraging me to undertake a PhD and giving me the opportunity to do so at CQSE GmbH. I also want to thank him for our productive discussions.

I wish to thank Oscar Luis Vera-Pérez for his interest in my work, his implementation of the Descartes plug-in, and our discussions.

I am grateful to the numerous people who collaborated on papers, reviewed papers, or gave feedback on this dissertation. They are, in alphabetical order: Florian Deißeböck, Nils Göde, Roman Haas, Elmar Juergens, Martin

Monperrus, Dennis Pagano, Jakob Rott, Tobias Röhm, Daniela Steidl, Alexander von Rhein, and Andreas Wilhelm. In addition, I am thankful to Florian Dreier and Jakob Rott for their work on the Test Impact Analysis and our discussions.

I want to thank my family for their invaluable support, not only during my dissertation, but throughout my education.

The funding from the German Federal Ministry of Education and Research (BMBF) is gratefully acknowledged.

Stuttgart, 7<sup>th</sup> October 2019

Rainer Niedermayr

# ZUSAMMENFASSUNG

Automatisierte Softwaretests sind eine wichtige Qualitätssicherungsmaßnahme in Softwareprojekten und helfen Fehler in einer Anwendung frühzeitig aufzudecken. Zur Bewertung von Testsuiten wurden in der Vergangenheit verschiedene Metriken und Verfahren vorgeschlagen. Dabei sind Code-Coverage-Metriken am weitesten verbreitet und werden vor allem in der kommerziellen Softwareentwicklung eingesetzt. Jedoch sind diese nur bedingt geeignet, die Effektivität von Testsuiten hinsichtlich ihrer Fehleraufdeckungsrate zu bewerten. Ein anderes wirkungsvolles und aussagekräftiges Verfahren ist Mutation Testing, bei dem Fehler in den Anwendungscode einer Software eingefügt werden und geprüft wird, ob die vorhandenen Testfälle diese aufdecken können. In Bezug auf die Bestimmung der Testeffektivität ist Mutation Testing anderen Verfahren deutlich überlegen, jedoch ist es sehr rechenintensiv und sogenannte äquivalente Mutanten können die Ergebnisse verfälschen. Wegen dieser Probleme wird Mutation Testing in der Praxis derzeit kaum eingesetzt.

Das Ziel dieser Dissertation ist es, aussagekräftigere Metriken und Verfahren zur Bewertung von Testsuiten zu entwickeln, welche mit vertretbarem Berechnungsaufwand anwendbar sind. Diese sollen Code-Coverage-Metriken in Bezug auf die Aussagekraft übertreffen und gleichzeitig weniger rechenintensiv als derzeit verwendete Mutation-Testing-Verfahren sein.

Dazu wurde ein leichtgewichtiges Verfahren zur Erkennung von schein-

getesteten Methoden konzipiert, umgesetzt und evaluiert. Scheingetestete Methoden sind von mindestens einem Testfall überdeckt, jedoch erkennt keiner der überdeckenden Testfälle, wenn die gesamte Logik aus der Methode entfernt wird. Außerdem wurde ein Machine-Learning-Modell zur Vorhersage von scheingetesteten Methoden entwickelt, welches ein neu eingeführtes Maß für die Aufrufdistanz zwischen Methoden und Testfällen sowie weitere kostengünstig berechenbare Metriken verwendet. Im Rahmen der Arbeit wurde ein weiteres Machine-Learning-Modell zur Identifizierung von Methoden mit einem niedrigem Fehlerrisiko vorgeschlagen. Solche Methoden können bei Qualitätssicherungsmaßnahmen nachrangig behandelt oder ausgeschlossen werden, sodass beispielsweise der Aufwand für die Erkennung scheingetesteter Methoden weiter reduziert wird.

Die Ergebnisse zeigen, dass scheingetestete Methoden in allen untersuchten Studienobjekten auftreten und relevante Testunzulänglichkeiten darstellen. Machine-Learning-Modelle können scheingetestete Methoden effizient vorhersagen, sodass diese Modelle als eine kostengünstige Annäherung vor einer Mutationsanalyse eingesetzt oder in Situationen verwendet werden können, in denen Mutationsanalysen nicht anwendbar sind. Mit einem weiteren Machine-Learning-Modell können basierend auf Code-Metriken Methoden identifiziert werden, die ein niedriges Fehlerrisiko aufweisen. Dies trifft auf etwa ein Drittel der Methoden zu, die folglich im Test mit einer niedrigeren Priorität behandelt werden können.

Die entworfenen Verfahren ermöglichen sowohl eine verhältnismäßig leichtgewichtige, anwendbare Berechnung von scheingetesteten Methoden als auch eine Vorhersage derselben basierend auf Methoden- und Testfallmetriken. Durch scheingetestete Methoden aufgedeckte Probleme in Testsuiten sind für Entwickler einfach verständlich und adressierbar, sodass Entwickler die Effektivität ihrer Testsuite verbessern können. Außerdem hilft die Identifikation von Methoden mit einem niedrigen Fehlerrisiko, Testaufwände auf relevante Methoden zu fokussieren. Effektivere Testsuiten können mehr Fehler bereits während des Softwareentwicklungsprozesses aufdecken und helfen damit, die Qualität eines Softwareprodukts zu verbessern sowie Fehlerfolgekosten zu reduzieren.

# ABSTRACT

Automated software tests are an important means of quality assurance in software projects and for helping to detect faults in software products early. While various measures and techniques have been proposed to evaluate test suites, code coverage metrics are the most common and are widely used in industry. However, it is questionable whether code coverage metrics are suitable to determine the fault detection capabilities of a test suite. Another powerful and valid technique for assessing test suites is mutation testing, which introduces faults into an application's code and checks whether the existing test cases can detect them. When determining fault detection capabilities, mutation testing is clearly superior to other measures, but it is computationally very complex and suffers from the problem of equivalent mutants, which distort the results. Due to these problems, mutation testing has rarely been adopted as a test adequacy criterion in practice.

The aim of this dissertation was to develop measures and techniques to better determine the effectiveness of a test suite with reasonable computational efforts. We wanted to come up with an approach that outperforms code coverage metrics in terms of validity and is, at the same time, less resource-intensive than currently used mutation testing approaches.

To do this, we proposed, implemented and evaluated a light-weight mutation approach to identify pseudo-tested methods; that is, methods that

are covered by at least one test case, but none of the test cases can detect the removal of the whole logic from the method. We further developed a machine-learning model to predict pseudo-tested methods based on a newly introduced measure for characterizing the proximity between methods and test cases and further easily computable measures. We also built machine-learning models to identify low-fault-risk methods, which can be excluded from quality-assurance activities to focus on the relevant methods and further speed up pseudo-testedness analyses.

The results show that pseudo-tested methods exist in all analyzed study objects and constitute relevant test inadequacies. Prediction models can efficiently identify pseudo-tested methods, which means that such models can be applied as a preceding, less costly step to mutation testing or be used in scenarios where mutation testing is not applicable. Depending on what level of risk is acceptable, approximately one-third of the methods can be classified as having a low fault risk, and these methods can be predicted with a machine-learning model based on code metrics.

The devised approaches make it possible to identify pseudo-tested methods by using an applicable, light-weight computation or a prediction based on method and test case metrics. Flaws in test suites uncovered by pseudo-tested methods are easy for developers to interpret and take action on, enabling the developers to improve the effectiveness of their test suite. The identification of methods with a low fault risk helps allocate test suite improvement efforts to the relevant methods. More effective test suites can reveal more faults during the software development phase, which can help improve the overall quality of a software product and reduce failure follow-up costs.



# CONTENTS

<b>1 Introduction</b>	<b>13</b>
1.1 Motivation . . . . .	14
1.2 Problem Statement . . . . .	15
1.3 Research Goal . . . . .	16
1.4 Contributions to Knowledge . . . . .	17
1.5 Previously Published Material . . . . .	19
1.6 Structure of Dissertation . . . . .	20
<b>2 Background</b>	<b>21</b>
2.1 Testing Terminology . . . . .	22
2.1.1 Test Cases and Test Suites . . . . .	22
2.1.2 Test Oracle . . . . .	23
2.1.3 Test Verdict . . . . .	24
2.1.4 Test Levels . . . . .	24
2.1.5 Test Tools . . . . .	26
2.1.6 Test and Application Code . . . . .	27
2.1.7 Regression Testing . . . . .	27
2.2 Error, Fault, Defect & Failure . . . . .	28
2.3 Code Coverage . . . . .	29

2.4 Mutation Testing . . . . .	30
2.4.1 Terminology . . . . .	31
2.4.2 Benefits and Downsides . . . . .	32
2.4.3 Tools . . . . .	34
2.5 Machine Learning . . . . .	36
2.5.1 Performance Evaluation . . . . .	36
2.5.2 Algorithms . . . . .	38
<b>3 State of the Art</b>	<b>41</b>
3.1 Code Coverage . . . . .	42
3.1.1 Studies Confirming Coverage as Effectiveness Indicator . . . . .	42
3.1.2 Studies Disproving Coverage as Effectiveness Indicator . . . . .	43
3.2 Checked Coverage & Assertion Coverage . . . . .	44
3.2.1 Checked Coverage . . . . .	45
3.2.2 Assertion Coverage . . . . .	45
3.2.3 Comparison . . . . .	46
3.3 Mutation Testing . . . . .	48
3.3.1 Addressing the Computational Complexity . . . . .	48
3.3.2 Addressing Equivalent Mutants . . . . .	51
3.3.3 Use In Practice . . . . .	51
3.3.4 Pseudo-Testedness . . . . .	53
3.4 Defect Prediction . . . . .	54
3.5 Test Case Selection and Prioritization . . . . .	56
3.6 Research Gap . . . . .	58
<b>4 Concept of Pseudo-Tested Methods</b>	<b>61</b>
4.1 Motivation . . . . .	62
4.2 Idea . . . . .	64
4.3 Definition of the Mutation Operator . . . . .	67
4.4 Implementation . . . . .	71
4.4.1 Configuration Options . . . . .	73
4.4.2 Alternative Implementation in Descartes . . . . .	74
4.5 Summary . . . . .	76

<b>5</b>	<b>Presence of Pseudo-Tested Methods and Their Characteristics</b>	<b>77</b>
5.1	Introduction	78
5.2	Study Objects	78
5.3	Execution of the Mutation Analysis	80
5.4	Existence of Pseudo-Tested Methods	80
5.5	Examples of Pseudo-Tested Methods	83
5.6	Characteristics of Pseudo-Tested Methods	88
5.7	Threats to Validity	96
5.8	Summary	99
<b>6</b>	<b>Predicting Pseudo-Tested Methods</b>	<b>101</b>
6.1	Motivation	102
6.2	Definitions	103
6.3	Approach	104
6.4	Study	107
6.4.1	Research Questions	107
6.4.2	Study Objects	108
6.4.3	Study Design	110
6.4.4	Data Collection and Processing	111
6.4.5	Results	114
6.4.6	Threats to Validity	121
6.5	Discussion	123
6.6	Summary	125
<b>7</b>	<b>Identifying Low-Fault-Risk Methods</b>	<b>127</b>
7.1	Motivation	128
7.2	Approach	130
7.2.1	Metric Computation	130
7.2.2	Data Pre-Processing	134
7.2.3	IDP Classifier	136
7.3	Study	137
7.3.1	Research Questions	137
7.3.2	Study Objects	138
7.3.3	Fault Data Extraction	139

7.3.4 Study Design . . . . .	140
7.3.5 Results . . . . .	145
7.3.6 Threats to Validity . . . . .	154
7.4 Discussion . . . . .	155
7.4.1 Limitations . . . . .	157
7.4.2 Applications . . . . .	157
7.4.3 Relation to Defect Prediction . . . . .	158
7.5 Summary . . . . .	159
<b>8 Enhanced Test Case Selection and Prioritization</b>	<b>161</b>
8.1 Motivation . . . . .	162
8.2 Existing Approach . . . . .	163
8.2.1 Required Input Data . . . . .	164
8.2.2 Test Case Selection . . . . .	164
8.2.3 Test Case Prioritization . . . . .	166
8.2.4 Provision of Results . . . . .	167
8.3 Enhanced Approach with Test Effectiveness Information . . . . .	167
8.4 Evaluation . . . . .	169
8.5 Limitations . . . . .	171
8.6 Summary . . . . .	172
<b>9 Conclusion and Future Work</b>	<b>173</b>
9.1 Discussion . . . . .	174
9.2 Limitations . . . . .	175
9.3 Future Work . . . . .	176
<b>Bibliography</b>	<b>179</b>
<b>List of Figures</b>	<b>209</b>
<b>List of Tables</b>	<b>211</b>
<b>List of Listings</b>	<b>213</b>
<b>List of Abbreviations</b>	<b>215</b>

# INTRODUCTION

*This chapter describes the motivation, problem statement, research goal, and contributions of this dissertation. It also presents the list of publications and outlines the structure of this dissertation.*

## 1.1 Motivation

Automated software tests are an important means of quality assurance in software projects. These tests execute parts of an application, compare the computed result or observed behavior with the expected one, and determine whether an application is acting as expected. Thus, they help reveal faults at an early stage and prevent regressions in software applications.

A significant proportion of software development efforts are spent on testing (Brooks Jr., 1995; Dustin *et al.*, 2009). Therefore, efforts should be made to use testing resources efficiently to maximize their benefit. In order to be able to decide which parts of an application benefit most from further testing and which ones can be considered as having been tested thoroughly enough, it is important to have a good understanding of an application's testing state. Therefore, a valid and expressive evaluation of a test suite's effectiveness is needed.

Various measures and techniques to evaluate test suites have been proposed. Code coverage metrics are the most common measures (Huang, 1975; Zhu *et al.*, 1997) and are widely used in industry (Yang *et al.*, 2009). Code coverage expresses which proportion of the application code is executed by test cases and can be computed at different levels. For example, line coverage expresses which proportion of an application's executable lines out of all lines are executed by tests. Common coverage metrics at other levels are branch coverage and decision coverage (Chilenski and Miller, 1994); they are relatively easy to compute and their performance overhead is negligible in the test suites of most applications. However, code coverage metrics measure test completeness and do not assess oracle quality. Therefore, they are not necessarily suitable for evaluating a test suite's effectiveness in terms of detecting faults (Antinyan *et al.*, 2018; Inozemtseva and Holmes, 2014).

More advanced approaches to evaluate test suites take data-flow criteria into account and check the relations between variable definitions and their uses (Rapps and Weyuker, 1982). For example, the all-defs criterion requires that every variable definition is executed and used at least once by a test case. Similarly, the concept of checked coverage proposed by Schuler and

Zeller (2011) measures the proportion of executable lines that is checked by test assertions. Those authors used dynamic slicing for computing the metric and found that it comes with a substantial run-time overhead and input/output (I/O) overhead for writing trace files. Besenreuther (2014) introduced assertion coverage, which is similar to checked coverage, but based on executed lines instead of executable ones. His approach traces the variables at run-time during the test execution. It comes with no I/O overhead for writing trace files, but has higher requirements regarding main memory than dynamic slicing. These approaches provide a more meaningful test evaluation regarding fault detection capabilities than code coverage, but are substantially more expensive to compute.

Another powerful technique with which to evaluate test suites is mutation testing (Jia and Harman, 2011), which involves generating mutants by seeding faults into the application code and checking whether the tests can kill (detect) these faults. This technique takes oracle quality into account and measures fault detection capabilities. While it provides more meaningful results than code coverage metrics, the increase in validity brings downsides in terms of execution time and applicability. Commonly used mutation operators create many mutants for a single method; in the worst case, all test cases need to be executed for each mutant to determine whether it can be killed or not. Hence, despite several optimization techniques, mutation testing is computationally complex due to the effort needed to generate and test a large number of mutants. Furthermore, equivalent mutants distort the results and are considered a major problem. Consequently, there are no indications that mutation testing has been widely adopted as a test efficacy criterion in practice (Ivanković *et al.*, 2018; Jia and Harman, 2011; Madeyski *et al.*, 2014).

## 1.2 Problem Statement

It is important to know how thoroughly a software application has been tested in order to decide whether a new version of an application is ready

for release or whether further testing is necessary to ensure that it behaves as expected. In the latter case, it is relevant to know which parts of the application have been insufficiently tested and would benefit most from further testing.

To answer these questions, code coverage metrics are the de facto standard in industry. However, a test case that triggers the execution of a large portion of an application contributes a lot of coverage, even if it does not contain any assertions. That is, these metrics do not take into account whether covered methods are tested with appropriate assertions, or whether they are executed without any assertions or only incomplete ones. According to [Fowler \(2004\)](#), test cases that do not contain any assertions are useless, unless their purpose is to check the absence of thrown exceptions. Consequently, the execution of code does not make it possible to conclude that the covered code has been tested effectively.

Therefore, more profound techniques are needed to determine the effectiveness of a test suite in terms of fault detection capabilities. Such techniques exist (for example, mutation testing, or checked coverage) and provide more meaningful insights into test effectiveness. However, their applicability in practice is limited, for various reasons. Therefore, there is a need to strike a balance between easily computable but less valid techniques and more valid but overly complex techniques.

### 1.3 Research Goal

The aim of this dissertation was to develop measures and techniques to better determine the effectiveness of a test suite with reasonable computational efforts. We wanted to come up with an approach that outperforms code coverage metrics in terms of validity and is, at the same time, less resource-intensive than currently used mutation testing approaches.

We also sought to study what characteristics point to methods that exhibit a low fault risk and can therefore be deferred or excluded in quality assurance (QA) activities. Furthermore, we wanted to investigate how test case selection



and prioritization approaches can be extended to further reduce the time to the first failure in a test suite execution, which would enable developers to investigate failures earlier.

Our research focused on automated software tests in test suites that were manually implemented by developers. Automated tests are critical for continuous integration ([Hilton \*et al.\*, 2017](#)) and are becoming ever more important. We studied test suites of libraries and systems implemented in Java and investigated them in respect to their capabilities to detect faults. We focused on Java, because it is among the most popular programming languages ([Chan, 2019](#); [Diakopoulos and Cass, 2016](#); [StackOverflow, 2019](#)). Java belongs to the object-oriented languages and is strongly typed ([Gosling \*et al.\*, 2013](#)). We assumed that our findings may also apply to other languages that exhibit these characteristics such as, for example, C#.

## 1.4 Contributions to Knowledge

This dissertation makes the following contributions to knowledge.

### Conceptual

- We present and define a novel mutation operator at the method level.<sup>1</sup> The purpose is to identify pseudo-tested methods, which are methods that are covered by at least one test case, none of which can detect when the whole logic is removed from the method.
- We define the minimal stack distance as a measure in the context of tests to quantify the proximity between methods and test cases.
- We define a set of metrics and create a machine-learning model to predict pseudo-tested methods without the need to execute a costly mutation analysis.
- We create a machine-learning model based on source-code metrics to identify methods whose code can be considered to be too trivial to test because they contain hardly any faults.

---

<sup>1</sup> I presented a preliminary version of this mutation operator in [Niedermayr \(2013\)](#).

- We devise a hybrid test case selection and prioritization approach that additionally incorporates test effectiveness information to reduce the time to the potentially existing first test failure in a test suite execution.

## Evaluations

- We conducted a mutation analysis with the presented mutation operator on 19 study objects and showed that they all exhibit pseudo-tested methods.
- We studied the characteristics of pseudo-tested methods and showed that they are relevant and should be tested more thoroughly. We also identified measures that exhibit a moderate correlation with pseudo-testedness in some study objects.
- We conducted an empirical study with 21 study objects and showed that our machine-learning model can successfully predict pseudo-tested methods.
- We conducted an empirical study with six study objects and showed that methods exist that are too trivial to test. We also presented the performance of machine-learning classifiers to predict these methods.

## Implementations

- We developed a new mutation analysis tool, which implements the presented mutation operator to identify pseudo-tested methods.
- We extended the existing mutation testing tool *Pitest (PIT)* by a new feature to compute a full mutation matrix and submitted it as a pull request, which was merged into the main branch.
- We implemented a dynamic analysis to compute the minimal stack distance measure between the methods and the test cases.
- We extended the hybrid test case selection and prioritization approach in the software-quality analysis suite *Teamscale* such that it additionally considers test effectiveness information.

## 1.5 Previously Published Material

Parts of the contributions presented in this thesis have been published in:

- Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. *Will My Tests Tell Me If I Break This Code?* Published in: Proceedings of the 1<sup>st</sup> International Workshop on Continuous Software Evolution and Delivery (CSED'16). ACM, 2016.
- Jakob Rott, Rainer Niedermayr,<sup>1</sup> Elmar Juergens, and Dennis Pagano. *Ticket Coverage: Putting Test Coverage into Context*. Published in: Proceedings of the 8<sup>th</sup> Workshop on Emerging Trends in Software Metrics (WETSoM'17). IEEE, 2017.
- Rainer Niedermayr, Tobias Röhm, and Stefan Wagner. *Poster: Identification of Methods with Low Fault Risk*. Published in: Proceedings of the 40<sup>th</sup> International Conference on Software Engineering Companion (ICSE'18 Companion). ACM, 2018.
- Rainer Niedermayr and Stefan Wagner. *Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?* Published in: Proceedings of the 23<sup>rd</sup> International Conference on Evaluation and Assessment in Software Engineering (EASE'19). ACM, 2019.
- Rainer Niedermayr, Tobias Röhm, and Stefan Wagner. *Too Trivial To Test? An Inverse View on Defect Prediction to Identify Methods with Low Fault Risk*. Published in: PeerJ Computer Science (5). PeerJ, 2019.
- Roman Haas, Rainer Niedermayr,<sup>2</sup> and Elmar Juergens. *Teamscale: Tackle Technical Debt and Control the Quality of Your Software*. Published in: Proceedings of the 2<sup>nd</sup> International Conference on Technical Debt (TechDebt'19 Tools Track). IEEE, 2019.

---

<sup>1</sup> I authored and reviewed drafts of the paper and approved the final draft.

<sup>2</sup> I reviewed drafts of the paper and approved the final draft.

## 1.6 Structure of Dissertation

The remainder of this dissertation is organized as follows:

- Chapter 2 provides the background on the main topics of this dissertation. It presents an overview of the testing terminology, describes existing techniques for evaluating test suites, and explains used machine-learning techniques.
- Chapter 3 presents the state of the art in this research area. It discusses related work regarding test effectiveness, defect prediction, and test case selection and prioritization.
- Chapter 4 introduces the concept of pseudo-testedness, describing the idea and its advantages, and drafting its realization.
- Chapter 5 describes pseudo-tested methods and their characteristics, explains that they are present in many projects, and shows their relevance.
- Chapter 6 introduces a measure to describe the proximity between test cases and methods. It presents a machine-learning classifier, which uses this proximity measure along with further easily computable measures, and shows that pseudo-tested methods can successfully be predicted without mutation analysis.
- Chapter 7 describes how methods can be identified that do not necessarily need to be tested due to a low fault risk. These methods can be skipped from a mutation analysis to make it even faster and its results more relevant.
- Chapter 8 shows how test case selection and prioritization approaches can be enhanced by including information about test effectiveness.
- Chapter 9 summarizes and discusses this dissertation and its limitations. It also discusses which future work could be undertaken on the basis of the completed work.

CHAPTER



## BACKGROUND

*This chapter provides an overview of the background to this dissertation. It introduces definitions for tests and their types, defines faults and related terms, describes code coverage metrics and mutation testing, and explains machine-learning techniques used in this dissertation.*

## 2.1 Testing Terminology

This section defines the testing terminology used in this dissertation. Figure 2.1 presents an overview of the most important terms and their relations.

### 2.1.1 Test Cases and Test Suites

A *test case* (also referred to as *test*) executes functions of a system (or a library) and compares the computed result or the observed behavior with the expected one. A test case will pass if the system behaves as expected; otherwise, it will fail (Zelkowitz, 2003). Thereby, a test case aims to detect faults and ensure the correctness of the implementation. A test case implicitly also checks the absence of thrown exceptions for a given program flow (Fowler, 2004).

A test case usually consists of actions and preconditions to prepare the system under test, actions and input data to trigger the execution of the intended computations, an oracle to determine the test verdict by validating computed results and observed behavior, and actions for the tear-down (Zhi and Garousi, 2013).

A test case can be either automated or manual. An *automated test case* is defined in the source code as a sequence of method calls and assertions, or specified as steps and conditions in other formats that can be interpreted by a test runner. Such a test case can be executed without human interaction (Taipale *et al.*, 2011). Automated test cases can either be implemented by developers or generated automatically from the source code using corresponding tools (Fraser and Arcuri, 2011). A *manual test case*, by contrast, is performed by a human (Taipale *et al.*, 2011) and may be based on a semi-formal test case description. This dissertation focuses solely on automated software tests that were created by developers.

A *test suite* is “a set of several test cases for a component or a system under test” (Homès, 2013). In object-oriented programming languages, test cases defined in the code are enclosed by a *test class*, which groups semantically

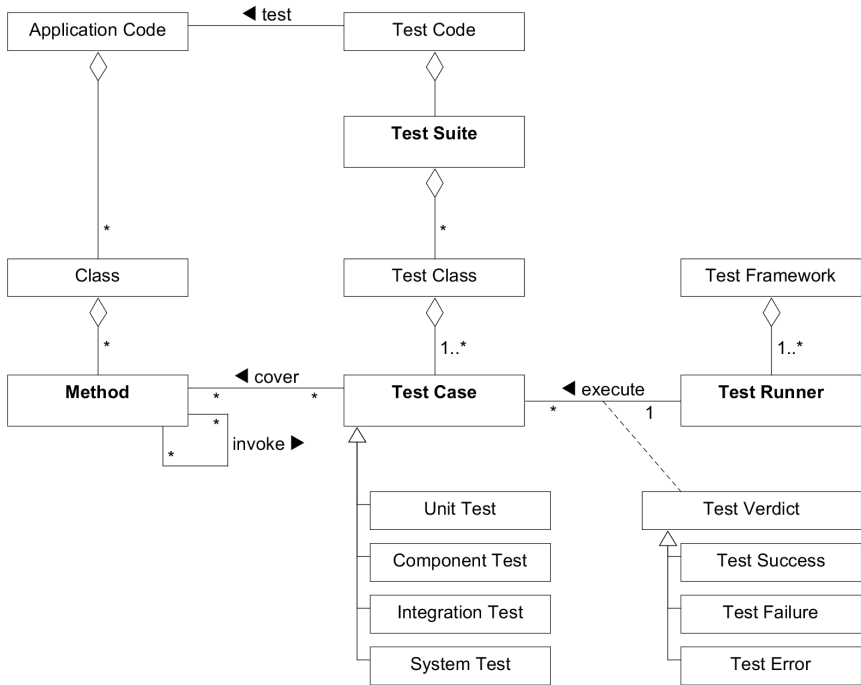


Figure 2.1: Overview of the testing terminology of automated tests.

related test cases and allows set-up procedures to be shared.

A *parameterized test* is a special variant of an automated test case that is specified once in the code and executed multiple times with different input parameters (Tillmann and Schulte, 2005).

### 2.1.2 Test Oracle

A *test oracle* determines whether a test passes or not (Barr et al., 2015; Naik and Tripathy, 2008). An automated test's oracle usually consists of assertions (Li and Offutt, 2017). A *test assertion* (also referred to as *assertion*) is a means of comparing an expected value with a computed value or observed behavior (Green and Kostovarov, 2013). Assertions in the form of method invoca-

tions are most common (e.g., `assertEquals` or `assertNotNull`). Deviation between an expected and an actual value results in a test failure.

### 2.1.3 Test Verdict

The *test verdict* is the outcome of the execution of a test case. A test case's verdict is *successful* if the test case passes without violating any assertions, without causing any (unexpected) exceptions or errors, and without exceeding a possibly specified time limit. Otherwise, the verdict of an executed test case is *failed*. Some test frameworks, for example, JUnit and TestNG, further differentiate between a test failure, which is caused by a violated assertion, and a test error, which is caused by an exception or an error thrown in the code under test (Langr *et al.*, 2015). We make no distinction between test failure and test error in the following. Depending on the test framework, further possible test verdicts may exist. For example, a test case that is known to be failing or flickering may be temporarily disabled by a developer so that it is not executed. It would still appear in the execution report of the test suite and its verdict would be *ignored*. Ignored test cases are not relevant within the context of this dissertation and are considered as if they were not present.

### 2.1.4 Test Levels

Test cases may be categorized based on their intended scope and the thereof derived goal. A *unit test* examines a small unit of code in isolation (Runeson, 2006); that is, it usually focuses on a single method or a class and its execution does not involve databases or other systems. The goal of a unit test is to ensure that the code under test behaves correctly in isolation (Elbaum *et al.*, 2006). Since a unit test executes only a small amount of code, its execution is fast and the effort to examine a failure and localize the underlying fault is low (Osherove, 2009). A *component test* deals with the next-largest scope and examines a component (module) of a system in isolation (Jin and Offutt, 1998). Next, an *integration test* tests the interplay among different compo-



nents or systems and aims to identify problems at their boundaries (Beizer, 1990); that is, it ensures that the modules communicate correctly based on the same assumptions. A *system test* examines a whole, integrated software system (Jin and Offutt, 1998), which may consist of several components. Such tests often trigger the execution of a large functionality and compare the end result (which can be, for example, aggregated data, a report, or a log file) with the expected one (Elbaum *et al.*, 2006). Unlike a unit test, a system test usually covers many methods of a system, and thus, it executes a lot of code. Consequently, a system test's execution time is relatively long. In addition, changes to any of the covered code may require adaptations in the test case, resulting in considerable maintenance efforts. Higher-level tests are also more prone to non-determinism problems (Fowler, 2012b). Furthermore, the effort involved in debugging a system test and localizing the cause of a test failure may also be significant due to the amount of covered code and the involvement of different components. This also applies to tests that are performed through the user interface (Vocke, 2018). They require especially high maintenance efforts because “graphical user interfaces tend to change frequently and test scripts have to be adapted to these changes” (Berner *et al.*, 2005).

In practice, “the distinction among integration testing, system testing, and user acceptance testing often blurs” (Li and Offutt, 2017). It is common, especially in open-source projects, that developers do not categorize automated tests at all and implicitly consider all tests as unit tests or distinguish only between unit tests and higher-level tests. The well-known build tool Maven<sup>1</sup> provides only two execution phases for tests by default: one for unit tests and one for integration tests (ASF, 2019).

Figure 2.2 presents the concept of the *test pyramid*, which was originally proposed by Cohn (2010). The test pyramid sketches the desired extent of test levels in terms of the number of tests and required execution time. It expresses that the number of tests at lower levels, such as, unit tests, should be proportionally high because these tests are small, fast, and relatively easy

---

<sup>1</sup> <https://maven.apache.org>

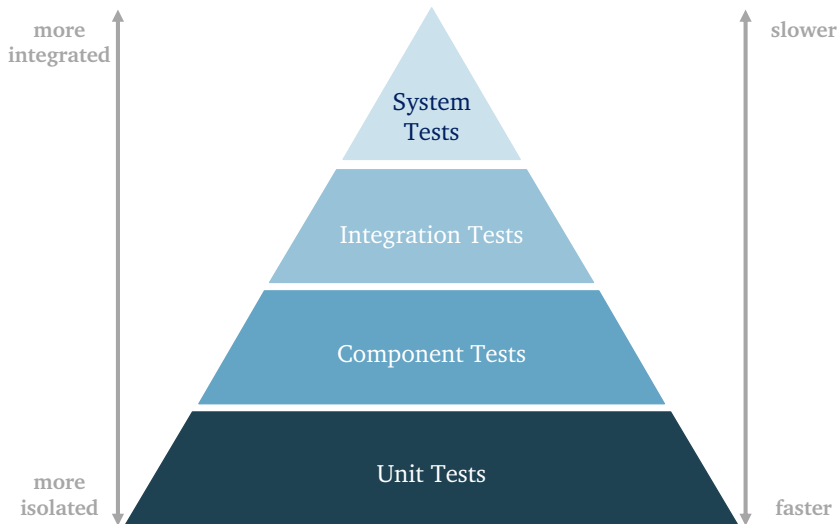


Figure 2.2: The test pyramid (following [Cohn \(2010\)](#)).

to analyze in case of failures. By contrast, higher levels of the pyramid, such as, integration or system tests, should contain proportionally fewer tests ([Fowler, 2012b](#); [Wacker, 2015](#)). Tests at higher levels are useful because they can detect integration problems that cannot be identified by unit tests, but they are much more expensive in terms of execution time, required maintenance, and debugging efforts.

### 2.1.5 Test Tools

A *test framework* is a software tool to support the writing and running of automated test cases ([Hamill, 2004](#)). It comprises functionality to execute tests and report the results. Well-known test frameworks for Java are *JUnit*<sup>1</sup> and *TestNG*.<sup>2</sup> They are primarily intended for unit tests, but they can be and are commonly used to execute tests of other levels as well. Examples of Java

---

<sup>1</sup> <https://junit.org>

<sup>2</sup> <https://testng.org/doc/index.html>

test frameworks targeting higher levels of the test pyramid are *JBehave*,<sup>1</sup> *FitNesse*,<sup>2</sup> *Citrus*,<sup>3</sup> and *Selenium*.<sup>4</sup> Further libraries support and facilitate the development of automated tests by providing additional assertion methods (e.g., *Hamcrest*,<sup>5</sup> *assertJ*,<sup>6</sup> and others), or by providing mechanisms to create test doubles in the form of mock objects (e.g., *Mockito*,<sup>7</sup> *JMockit*,<sup>8</sup> and others).

### 2.1.6 Test and Application Code

We define *test code* as the part of the code of a system (or a library) that is used solely for testing and is not included in releases deployed to production environments. Test code comprises preparation methods executed before the tests (to initialize test data and set up environments), test cases (to perform the tests), tear-down methods executed after the tests (to reset the state), and further utilities to support testing (e.g., custom assertion methods). By contrast, we define *application code* as the code of a system (or a library) that is running in production. The application code does not comprise experimental or sample code, does not overlap with the test code, and is the subject that is tested by the test cases. Code coverage is measured based on the application code.

### 2.1.7 Regression Testing

*Regression testing* is a testing activity to “provide confidence that newly introduced changes do not obstruct the behaviors of the existing, unchanged part of the software” (Yoo and Harman, 2012). Hence, it aims to detect regression faults that are introduced during the evolution of a system.

---

<sup>1</sup> <https://jbehave.org>

<sup>2</sup> <http://docs.fitnesse.org>

<sup>3</sup> <https://citrusframework.org>

<sup>4</sup> <https://www.seleniumhq.org>

<sup>5</sup> <http://hamcrest.org/JavaHamcrest>

<sup>6</sup> <https://joel-costigliola.github.io/assertj>

<sup>7</sup> <https://site.mockito.org>

<sup>8</sup> <https://jmockit.github.io>



Figure 2.3: Relation between fault, defect, and failure.

## 2.2 Error, Fault, Defect & Failure

An *error* is a “human action that produces an incorrect result” (IEEE, 2010) and a *fault* is a “manifestation of an error in software” (IEEE, 2010). Hence, a fault is an incorrect step or data definition in the code of a software. A *regression fault* (or *software regression*) corresponds to a fault in functionality from a previous version of a software and is caused by bugfixes and new functionality in a newer software version (Agrawal *et al.*, 1993a). A *defect* is a fault that is detected during the use of a released software product (IEEE, 2010); that is, it is detected after the implementation and testing phase. A fault may cause failures. A *failure* is an “event in which a system or system component does not perform a required function within specified limits” (IEEE, 2010); hence, a failure is a deviation from the specified behavior. Figure 2.3 illustrates the relation among fault, defect, and failure.

The *RIPR model* by Li and Offutt (2017) defines four criteria that are necessary for a failure to be observed. It is an extension of the RIP model, which was independently developed by Morell (1984) and Offutt (1988) in the 1980s. To be able to detect a fault, the fault needs to be reached; that is, be executed by a test case (*reachability*). Next, the execution of the faulty location must cause an incorrect program state (*infection*). Furthermore, the infected state must propagate to an incorrect final state or output (*propagation*). Finally, a tester or a test oracle (*e.g.*, an assertion) must observe the incorrect part of the final state or output (*revealability*).

## 2.3 Code Coverage

Code coverage is a metric that expresses which proportion of application code of a software project is executed when running all test cases (Zhu *et al.*, 1997). It can be computed at different levels. *Method coverage* (or *function coverage*) corresponds to the proportion of methods that are (partially or completely) executed by test cases out of all methods of the application code. *Statement coverage* measures which proportion of the executable statements are invoked at least once during testing (Hayhurst *et al.*, 2001). It is the most used coverage criterion in practice (Mathur, 2013). Similarly, *line coverage* measures the proportion of executed lines out of all coverable lines (Tikir and Hollingsworth, 2002). *Branch coverage* measures the proportion of branches that are executed by the test cases (Homès, 2013). *Decision coverage* measures the proportion of decisions that are executed by the test cases (Homès, 2013), in which a decision is compound of one or more conditions. One hundred percent decision coverage requires that each decision is evaluated to both *true* and *false* (Hayhurst *et al.*, 2001). The difference to branch coverage results from the fact that a branch may not necessarily exist for both outcomes of a decision (*i.e.*, an else-branch may not be present). *Condition coverage* is more fine-grained and measures whether each condition in a decision is evaluated to both possible outcomes (Hayhurst *et al.*, 2001). *Modified condition/decision coverage (MC/DC)* measures “if every condition within a decision has taken on all possible outcomes at least once, and every condition has been shown to independently affect the decision’s outcome” (Rajan *et al.*, 2008). For highly critical software in the avionics industry, a test suite that satisfies full MC/DC coverage of the source code is required (RTCA, 1992). Finally, *path coverage* measures the proportion of the execution paths from an entry point to an exit point (*e.g.*, all paths within a method) that are executed during testing (Zhu *et al.*, 1997).

Coverage can also be applied at other levels outside the source code. For example, *ticket coverage* expresses which proportion of the methods added or modified during the implementation of a ticket (issue) have been tested (Rott *et al.*, 2017).

When we refer to the term *code coverage* in this dissertation, we mean coverage at the method level (unless specified otherwise). We use the following definitions.

Let  $M$  be the set of the methods of the application code and let  $T$  be the corresponding test suite.

We define a method  $m \in M$  as *covered* if the test suite  $T$  contains at least one test case  $t$  that executes at least one of the statements in the body of  $m$ .

We define the *test-case specific coverage* of a test case  $t \in T$  as the set of the methods that are covered by  $t$ .

We define the *overall coverage* of an application as the union of the test-case specific coverage of all test cases  $t \in T$ .

## 2.4 Mutation Testing

Mutation testing is an established technique to assess test suites (Offutt and Untch, 2001). It was first proposed by Lipton (1971) in the 1970s and formalized by DeMillo *et al.* (1978) and it has been studied extensively since then (Jia and Harman, 2011; Papadakis *et al.*, 2017; Usaola and Mateo, 2010). The general idea behind mutation testing is to generate mutants by introducing faults into a program and checking if the tests can detect (*kill*) the faults.

Mutation testing is based on two assumptions: the *competent programmer hypothesis* and the *coupling effect hypothesis* (Zhu *et al.*, 1997). According to the competent programmer hypothesis of DeMillo *et al.* (1978), programmers are competent and tend to “create programs that are close to being correct.” In practice, this hypothesis may not be valid for large programs, but it suffices for mutation testing that the hypothesis holds with respect to an individual fault (Gopinath *et al.*, 2014b). The consequence of the competent programmer hypothesis is that most faults are small and can be corrected by a few syntactical changes (Jia and Harman, 2011). The coupling effect claims that complex faults are coupled to simple faults in such a way that a test suite

capable of detecting small faults is sensitive enough to detect more complex faults (DeMillo *et al.*, 1978). The coupling effect has been investigated theoretically by Wah (2000) and Wah (2003) as well as empirically by Offutt (1992) and found to be valid.

### 2.4.1 Terminology

A *mutation operator* is a deterministic transformation rule that generates a *mutant* by applying syntactical changes to the original program (King and Offutt, 1991). For example, a mutation operator may replace an addition with a subtraction, negate a condition, or remove a statement (as long as the code remains syntactically valid; that is, compilable).

Mutations can be performed on the source code or on the byte code. Modern mutation testing tools operate on the byte code to avoid the need for a compilation, which is slow and requires access to libraries that are declared as dependencies.

After a mutant is created, test cases are executed to check whether they can detect the fault, which was introduced by the mutation. State-of-the-art mutation testing tools run all test cases once before the actual mutation analysis and record which test case covers which method. This enables a tool to select the relevant test cases to examine a mutant so that the test execution can be limited to test cases that cover the altered code.

A mutant is said to be *killed* if at least one test case of the test suite fails due to the changes; otherwise it is said to have *survived* or be *alive* (Zhu *et al.*, 1997). A mutation may be killed by a test case by a failing assertion in the test or by a failing implicit check in the source code (*e.g.*, a `NullPointerException`, or an `ArithmeticException` caused by a division through zero, or a thrown `IllegalArgumentException`).

An *equivalent mutant* is—despite syntactical changes—semantically equivalent to the original program such that no possible change in behavior can be observed (Jia and Harman, 2011). Consequently, such a mutant does not represent an injected fault and cannot be killed by any test case; Listing 2.1 presents an example. Equivalent mutants cannot be detected automatically

because the equivalence of two functions is generally not decidable (Budd and Angluin, 1982).

```
1 public int computeFactorial(int n) {
2     int result = 1;
3     for (int i = n; i >= 1; i--) {
4         result = result * i;
5     }
6     return result;
7 }
```

Listing 2.1: Example of an equivalent mutant. A mutation operator like PIT’s “Conditionals Boundary Mutator,” which replaces  $>$  with  $\geq$ , will create an equivalent mutant in this method. Although the body of the loop will be executed once more often after the mutation, the outcome will not be influenced by multiplying `result` with one in the last pass.

The outcome of a mutation analysis is a list of the mutations with their test verdict. A further aggregated measure is the *mutation score* (or *mutation adequacy score*), which corresponds to the proportion of killed mutants out of all created mutants (Jia and Harman, 2011). A test suite is considered *mutation adequate* if its mutation score achieves 100% (Offutt *et al.*, 1996a).

#### 2.4.2 Benefits and Downsides

By inserting faults into a program and checking whether test cases can detect them, mutation testing assesses the fault detection capabilities of a test suite. Andrews *et al.* (2005), Frankl *et al.* (1997), and Just *et al.* (2014a) provided empirical evidence for that and showed that mutation testing outperforms coverage-based metrics as an indicator of test effectiveness.

However, mutation testing suffers from two major drawbacks, which is why it is not widely used in practice (Ivanković *et al.*, 2018; Jia and Harman, 2011; Zhu *et al.*, 2018). First, mutation testing is computationally complex. Many mutants may be created for each method and, in the worst case, all test



cases need to be executed for each mutant to determine whether a mutant can be killed or not.

Second, equivalent mutants are a problem because they cannot be detected automatically (Offutt and Pan, 1997) and distort the results. Grün *et al.* (2009) identified equivalent mutants as an important problem because they are surprisingly common. Equivalent mutants lower the mutation score and force developers to needlessly analyze mutants in order to figure out which part of the code needs to be tested more thoroughly and how this can be achieved. According to Schuler and Zeller (2013b), a developer requires an average of 15 minutes to determine whether a mutant is equivalent or not. Similarly, Ivanković *et al.* (2018) reported a duration of 11.7 minutes per mutant. Thereby, equivalent mutants threaten the developer's acceptance of mutation analysis results.

A further obstacle to a wider use in industry is the mutation testing tools' lack of integration with software development infrastructures and processes (Madeyski *et al.*, 2014). In particular, there is no clear methodology for using the results of a mutation analysis. Most developers will ignore the results if they are not presented in an appealing way in tools that the developers already use in their daily work. Furthermore, the mutation score as such is not actionable and the sheer number of mutants overwhelms developers. In order to be beneficial for developers, mutation testing tools need to be executed within continuous integration, focus on recently changed code, present findings in an understandable way in an established tool, and ideally give hints on how to fix the problems.

Petrović and Ivanković (2018) reported about the integration of mutation testing into Google's development workflow. At Google, an incremental mutation analysis is triggered when a pull request is created. A limited number of mutants focusing on changed code are presented to the developers in the code review tool (see Section 3.3.3).

### 2.4.3 Tools

Many mutation testing tools have been developed for various programming languages. Commonly known tools for the programming language Java are *Pitest* (*PIT*),<sup>1</sup> *Javalanche*,<sup>2</sup> *Major*,<sup>3</sup> *Judy*,<sup>4</sup> and *MuJava*.<sup>5</sup>

The focus in this section is on *PIT* because it is the best-known and most mature mutation testing tool for Java applications (Delahaye and Du Bousquet, 2013; Gopinath *et al.*, 2017). Its code is open-source and maintained by an active community (Coles *et al.*, 2016). Furthermore, it integrates with build tools, such as, Maven and Gradle. *PIT* has been used in several empirical software engineering studies (e.g., in Ahmed *et al.* (2016), Gopinath *et al.* (2015), and Gopinath *et al.* (2016)).

The mutation analysis in *PIT* works as follows. First, *PIT* scans all classes in the system under test to identify possible mutation points and stores their location along with the intended mutation operator. Next, *PIT* runs all tests once to perform a line coverage analysis so that it knows the test-case specific coverage of each test at the line level. It also records the duration of each test case. *PIT* then analyzes all mutants that are covered by at least one test case. To do so, based on the previously identified mutation points, the tool applies the intended mutation operator to the byte code in order to generate a mutant one after another (Coles *et al.*, 2016). The resulting mutant is kept in memory instead of writing it to the disk. Afterwards, a new Java Virtual Machine (JVM) is started for the test execution, and covering test cases are executed against the mutant. The test cases are prioritized by a heuristic, which takes a test's execution speed, its line coverage, and naming conventions into account (Coles, 2019a). The analysis of a mutant aborts as soon as the first test case kills the mutant. A mutant may also be killed by a time-out or a memory error during testing; for example, when a mutation introduces an endless loop. Overall, *PIT* applies several optimizations to

---

<sup>1</sup> <http://pitest.org>

<sup>2</sup> <https://github.com/david-schuler/javalanche>

<sup>3</sup> <http://mutation-testing.org>

<sup>4</sup> <http://madeyski.e-informatyka.pl/tools/judy>

<sup>5</sup> <https://cs.gmu.edu/~offutt/mujava>

reduce the analysis duration; for example, it operates at the byte code, selects the tests to run against the mutants, and minimizes the number of mutant executions (Coles *et al.*, 2016).

The mutation operators used in PIT are designed to generate hard-to-kill mutants, which comprise a minimal number of equivalent mutants (Coles, 2019c). The seven operators that are enabled by default are:

- Conditionals Boundary Mutator: replaces  $<$  with  $\leq$ ,  $>$  with  $\geq$ , and vice versa
- Increments Mutator: replaces increments with decrements, and vice versa
- Invert Negatives Mutator: replaces the sign of numeric variables
- Math Mutator: replaces binary arithmetic operations (e.g.,  $*$  with  $/$ )
- Negate Conditionals Mutator: replaces conditional operators (e.g.,  $==$  with  $!=$ )
- Return Values Mutator: changes return values (e.g., dynamically returns 1 if the original return value is 0, or 0 otherwise)
- Void Method Calls Mutator: removes calls to void methods

Finally, PIT provides an incremental mutation analysis as an experimental feature (Coles, 2019b). PIT tracks changes to code and tests and keeps results from previous mutation analyses. This information is used in future analyses to infer the mutation testing verdict of mutants that are located in unchanged code so that it is not necessary to conduct any tests for them. The incremental analysis makes a few assumptions and optimizations, which “introduce a degree of potential error into the analysis” (Coles, 2019b). For example, to decide whether the behavior of a class has changed, PIT only considers changes to the byte code of this class, its super classes, and outer classes; it does not consider dependencies to other classes that interact with this class. In addition, changes to logic or data defined outside the Java code, such as, test data or code in other programming languages, are not tracked at all.

## 2.5 Machine Learning

Machine-learning algorithms analyze existing data with the goal of deriving mathematical models to make predictions or decisions (Kobayashi *et al.*, 2011). Below, we explain how predictive machine-learning classifiers can be evaluated and present two algorithms.

### 2.5.1 Performance Evaluation

Machine-learning models with a binary outcome are usually evaluated by computing their precision and recall. *Precision* expresses the proportion of correctly predicted items of a given label out of all items that were predicted with this label. Hence, precision addresses the purity in retrieval performance; in other words, it is “a measure of effectiveness in excluding nonrelevant items from the retrieved set” (Buckland and Gey, 1994). It is computed as  $\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$ . *Recall* expresses the proportion of correctly predicted items of a given label out of all items that actually have this label. Recall “can be viewed as a measure of effectiveness in including relevant items in the retrieved set” (Buckland and Gey, 1994) and is computed as  $\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$ . The *F-score* (also known as *F<sub>1</sub>-score* or *F-measure*) combines precision and recall and is computed as  $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$ .

To evaluate the performance of a machine-learning model, the underlying data set needs to be separated into training data, which is used to create the model during the learning phase, and test data, which is used to assess the model. The most common technique for that is *10-fold cross-validation* in which the data set is split into 10 disjunct partitions of (nearly) equal size (Arlot and Celisse, 2010). 9 partitions are used for training and the 10<sup>th</sup> partition is used for the performance evaluation (e.g., by computing precision and recall). This is repeated for each partition so that each one is used exactly once for testing. The performance results are then averaged over the 10 partitions. Figure 2.4 illustrates this technique.

*Cross-project validation* is another technique that is commonly used in software engineering studies to separate training and test data for evaluations.

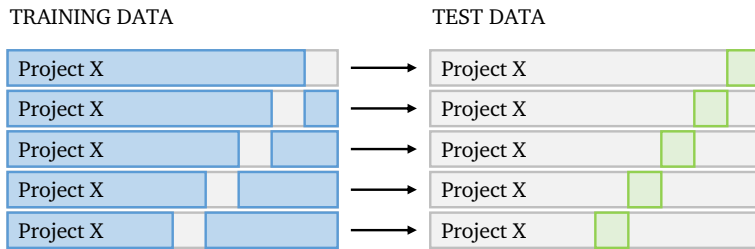


Figure 2.4: 10-fold cross-validation. The data of one project is split into 10 partitions. The classifier is trained on 9 partitions and tested on data of the 10<sup>th</sup> partition. This is repeated for all partitions.

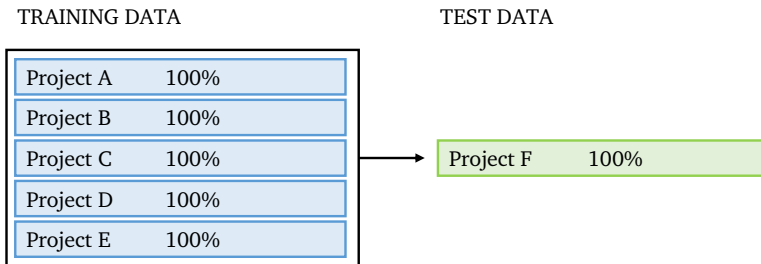


Figure 2.5: Cross-project validation. The machine-learning model is trained on all but one project, and evaluated against the remaining project. This is repeated for all projects.

A prediction model is trained on one or more projects (study objects) and tested on another unseen project. The aim of cross-project validation is to measure how generalizable a trained prediction model is; that is, how well it can be applied to other projects. This is relevant when no (precise) training data is available to create a project-specific model, as it is often the case in defect prediction. Generalizable models from other projects or companies can be an alternative in this scenario (Zimmermann *et al.*, 2009). Figure 2.5 illustrates the cross-project validation technique.

## 2.5.2 Algorithms

In this dissertation, we mainly use *Random Forest* and *Association Rule Mining* when we employ machine-learning techniques.

### 2.5.2.1 Random Forest

Random forest is a supervised machine-learning algorithm for classification and regression, and was proposed by [Breiman \(2001\)](#). Random forest belongs to the ensemble learning methods based on decision trees; that is, the algorithm generates many independent decision trees as predictors and aggregates their results ([Liaw and Wiener, 2002](#)). [Dietterich \(2000\)](#) found that ensemble classifiers are often more accurate than any of the individual classifiers that form an ensemble.

A random forest classifier creates a large number of independent trees and adds different layers of randomness to each tree. For each tree, a random sample of the training data is selected. Then, at each node of the tree, a subset of the variables (features) is randomly selected. The tree node is split using the best among the considered variables, unlike in standard trees where the best variable out of all variables is used for the split ([Liaw and Wiener, 2002](#)). To determine the best split, the *Gini Index* is commonly used, which measures variable impurity with respect to misclassifications ([Tat, 2017](#)). Each tree grows to the maximum depth using a combination of variables and is not pruned ([Pal, 2005](#)). The number of generated trees and the number of variables considered at each tree node is parameterized. When performing classification of new data, the prediction results of the different trees are aggregated by applying unweighted majority voting ([Liaw and Wiener, 2002](#)). The random forest classifier also supports regression; in this case, prediction results are aggregated by computing the average ([Liaw and Wiener, 2002](#)).

Random forest classifiers do not overfit with an increasing number of trees ([Liaw and Wiener, 2002](#)) because of the law of large numbers ([Feller, 1968](#)). They can handle unbalanced data as well as incomplete data with missing

values (Pal, 2005), and support categorical data (Pal, 2005). Furthermore, random forest classifiers provide an estimation of a variable's relative importance by permuting data for a variable and observing the increase in prediction error (Liaw and Wiener, 2002). The computation of a random forest model can easily be parallelized (Breiman, 2001) because the model's individual trees are independent.

### 2.5.2.2 Association Rule Mining

Association rule mining is a machine-learning technique for identifying relations between variables in a large dataset. It was introduced by Agrawal *et al.* (1993b). A dataset contains *transactions* consisting of a set of *items* that are binary attributes. An *association rule* represents a logical implication of the form  $\{ \textit{antecedent} \} \rightarrow \{ \textit{consequent} \}$  and expresses that the *consequent* is likely to apply if the *antecedent* applies. Antecedent and consequent both consist of a set of items and are disjoint. The *support* of a rule expresses the proportion of the transactions that contain both antecedent and consequent out of all transactions. The support of an item  $X$  with respect to all transactions  $T$  is defined as  $\textit{supp}(X) = \frac{|\{t \in T: X \subseteq t\}|}{|T|}$ . It is related to the significance of the itemset (Simon *et al.*, 2011). The *confidence* of a rule expresses the proportion of the transactions that contain both antecedent and consequent out of all transactions that contain the antecedent. The confidence of a rule  $X \rightarrow Y$  is defined as  $\textit{conf}(X \rightarrow Y) = \frac{\textit{supp}(X \cup Y)}{\textit{supp}(X)}$ . It can be considered as the precision (Simon *et al.*, 2011). A rule is *redundant* if a more general rule with the same or a higher confidence value exists (Bayardo *et al.*, 1999).

A major advantage of association rule mining is the natural comprehensibility of the rules (Simon *et al.*, 2011). Other commonly used machine-learning algorithms, such as support vector machines (SVM) or Naive Bayes classifiers, generate black-box models, which lack interpretability. Even decision trees can be difficult to interpret due to the subtree-replication problem (Simon *et al.*, 2011). Another advantage of association rule mining is that the gained rules implicitly extract high-order interactions among the predictors.

Association Rule Mining is primarily used for the market basket analyses

(Brin *et al.*, 1997). It has also been successfully applied in defect prediction studies (Czibula *et al.*, 2014; Ma *et al.*, 2010; Song *et al.*, 2006; Zafar *et al.*, 2012).



CHAPTER  
3

## STATE OF THE ART

*This chapter presents an overview of related work to this dissertation. It discusses related work in the areas test effectiveness, defect prediction, and test case selection and prioritization, and identifies research gaps.*

## 3.1 Code Coverage

Code coverage metrics measure the proportion of code that is executed by test cases. Although code coverage does not assess whether a test suite adequately checks the observed behavior, it has been proposed as an indicator of test effectiveness. Several studies have been conducted to investigate whether it is valid to use code coverage for that, by investigating whether coverage at different levels correlates with a test suite's fault detection capabilities. The studies provide conflicting evidence on this issue. [Schwartz \*et al.\* \(2018\)](#) suggested that the fault types used in the studies are one reason for the contradictory results because some types of faults are more difficult to detect than others.

Below, we briefly summarize studies that confirm or disprove that code coverage is a valid indicator of test effectiveness.

### 3.1.1 Studies Confirming Coverage as Effectiveness Indicator

An early study on the validity of coverage was conducted by [Wong \*et al.\* \(1994\)](#). They showed that the correlation between fault detection effectiveness and block coverage is higher than between effectiveness and size of the test set. They concluded that test cases that do not add additional coverage are likely ineffective in detecting faults. Similarly, [Hutchins \*et al.\* \(1994\)](#) showed that suites achieving high coverage levels are better at detecting faults than other randomly composed test suites of the same size. However, they remarked also that “100% code coverage alone is not a reliable indicator of the effectiveness.” To similar conclusions came [Chen \*et al.\* \(1995\)](#), [Del Frate \*et al.\* \(1995\)](#), [Frankl and Iakounenko \(1998\)](#), [Frankl \*et al.\* \(1997\)](#), and [Horgan \*et al.\* \(1994\)](#).

[Andrews \*et al.\* \(2006\)](#) conducted an empirical study on an industrial program with known faults. Their results showed that more demanding coverage criteria lead to larger test suites that detect more faults. Similarly, [Mockus \*et al.\* \(2009\)](#) revealed that an increase in coverage exponentially increases test efforts and linearly reduces field problems. They suggested

that “code coverage is a sensible and practical measure of test effectiveness.”

[Namin and Andrews \(2009\)](#) studied the relationship between size, coverage, and fault-finding effectiveness of test suites. They found that size and coverage are important for effectiveness and suggested a nonlinear relationship between them.

[Gligoric et al. \(2013\)](#) and [Gligoric et al. \(2015\)](#) conducted a study with Java and C programs. According to their results, branch coverage is the best predictor of mutation score and should be used to compare different test suites. By contrast, [Gopinath et al. \(2014a\)](#) determined that statement coverage is the best predictor of mutation kills, outperforming block, branch, and path coverage.

[Kochhar et al. \(2015\)](#) analyzed two large software systems with real bugs and found that statement and branch coverage correlate with the effectiveness in detecting bugs. The correlation was moderate in one project and strong in the other one.

More recently, [Bach et al. \(2017\)](#) studied on a large industrial SAP system whether covered code exhibits fewer future bugs than uncovered code. They analyzed 16,000 real bugs, found that the studied relationship is statistically significant, and regarded coverage “as a meaningful metric to estimate the adequacy of testing.”

### 3.1.2 Studies Disproving Coverage as Effectiveness Indicator

Several studies came to different conclusions. For example, [Briand and Pfahl \(1999\)](#) found no causal dependency between any coverage criterion and defect coverage.

Some years later, [Rajan et al. \(2008\)](#) investigated systems from the civil avionics domain. They showed that the “MC/DC metric is highly sensitive to the structure of the implementation and can therefore be misleading as a test adequacy criterion.” They also stated that “these criteria can easily be ‘cheated’ by restructuring a program to make it easier to achieve the desired coverage.”

[Wei et al. \(2010\)](#) conducted an empirical study in which they found that

“branch coverage is not a good indicator for the effectiveness of a test suite.” To similar conclusions came [Staats et al. \(2012\)](#) who investigated branch and MC/DC coverage criteria. They stated that both coverage criteria are only weak effectiveness indicators and therefore unsuitable for determining test suite adequacy.

A well-known study by [Inozemtseva and Holmes \(2014\)](#) on Java programs found that “high levels of coverage do not indicate that a test suite is effective.” Furthermore, [Inozemtseva and Holmes](#) discovered that the type of coverage had little effect on the correlation. Similarly, [Hemmati \(2015\)](#) suggested that statement coverage is a weak criterion, and [Ahmed et al. \(2016\)](#), [Kochhar et al. \(2017\)](#), and [Tengeri et al. \(2016\)](#) found that coverage is not a good predictor of the expected number of defects.

More recently, [Antinyan et al. \(2018\)](#) evaluated the adequacy of unit test coverage at Ericson. According to their results, the correlation of statement, decision, and function coverage with defects is weak. They concluded that “current unit test coverage measures do not seem to be any tangible help in producing defect-free software.”

Finally, several practitioners ([Benoit, 2014](#); [Burns, 2019](#); [Bytes, 2016](#); [Fowler, 2012a](#); [Lee, 2016](#); [Mansoor, 2016](#); [Marick, 1997](#); [Mols, 2017](#); [Seemann, 2015](#); [Skinner, 2010](#); [Xls, 2017](#); [Zimmermann, 2017](#)) critically discussed code coverage in blog posts. While they accepted that coverage can generally be useful for finding untested code, they all considered its widespread use as a quality target as highly questionable. They argued that code coverage is a misleading metric giving a false sense of confidence, and presented examples of that. They all suggested not to use it as a stipulated quality target.

## 3.2 Checked Coverage & Assertion Coverage

Checked coverage and assertion coverage are further coverage measures, which take test oracles into account and aim to measure a test suite’s fault detection capabilities.

### 3.2.1 Checked Coverage

Checked coverage was proposed by [Schuler and Zeller \(2011\)](#) and expresses which proportion of *executable* lines are checked by test assertions. That is, “statements that are executed, but whose outcomes are never checked, would be considered uncovered” ([Schuler and Zeller, 2013a](#)).

Their proposed implementation uses dynamic backward slicing. In the first step, the execution of each test case is traced and stored to a trace file. In the second step, a dynamic slice is created for each test assertion and statements are identified that influence the value checked in the assertion. The proportion of lines that are in at least one dynamic slice out of all executable lines constitutes the outcome.

### 3.2.2 Assertion Coverage

Assertion coverage is a similar measure proposed by [Besenreuther \(2014\)](#) and “indicates which *executed* lines contribute to values checked by test assertions.” His approach to compute this measure performs run-time tracing of test cases. It “builds the dependency graph during test execution, and the line executions done inside assertion functions or methods are used as a starting point for traversing the graph. All lines that occur in nodes that are reached during the traversal are considered assertion covered” ([Besenreuther, 2014](#)).

One challenge of this approach is the memory consumption, which can be excessive depending on the covered scope of a test case. It will be especially high when many values have a long life-time or are far apart from test assertions. In the study by [Besenreuther](#), up to 1.7% of the tests could not be analyzed due to the memory consumption. He states that “the share might be higher on more complex test suites.”

Another challenge regarding the implementation of such an approach are test cases invoking multi-threaded code. Each thread needs to be assigned to an individual tracer and tracers need to share the dependency graph and intermediate results among each other.

Furthermore, the outcome may become slightly inaccurate if certain parts of the analyzed code cannot be instrumented such that dependencies are missed. For example, native methods in Java code are not compiled to the byte code and cannot be instrumented.

### 3.2.3 Comparison

Both approaches aim to be more valid than traditional code coverage metrics and therefore take test oracles into account. Both are dynamic analyses and need to execute the whole test suite once.

They differ in their analyzed scope: checked coverage is based on *executable* lines, while assertion coverage is based on *executed* lines.

Furthermore, while checked coverage in the proposed form considers only assertions as oracles, assertion coverage additionally takes expected exceptions into account (Besenreuther, 2014). However, both approaches do not consider checks within the application code (e.g., parameter validations). This can be considered as problematic because Schuler and Zeller (2013a) showed that test cases from which all assertions are removed still kill over 50% of the mutants killed by the original tests.

The approaches also differentiate themselves regarding the design of the computation. While checked coverage uses dynamic backward slicing, assertion coverage applies run-time test tracing. “Dynamic slicing has a substantial run-time overhead compared to regular test execution” (Besenreuther, 2014) because writing the trace file is expensive due to the I/O and compression overhead. Therefore, different as expected, checked coverage does not outperform modern mutation testing tools in terms of performance (Schuler and Zeller, 2011). Assertion coverage, which “monitors the origin of all values and establishes all inter-dependencies in the code during test execution” (Besenreuther, 2014), is more efficient than checked coverage, however, it requires more main memory.

Finally, a drawback of both approaches compared to mutation testing is that a statement considered as “checked covered” respectively “assertion covered” may still contain faults. This is because assertions do not necessarily

uncover all faults. [Zhang and Mesbah \(2015\)](#) investigated assertion methods and found that there is a substantial difference between the effectiveness of different types. According to their results, `assertTrue/assertFalse` are more effective than assertions of type `assert(Not)Equals`, which are more effective than ones of type `assert(Not)NotNull`.

Listing 3.1 presents a code example in which a test case covers all statements of the method `createRectangle(...)` and performs an assertion on the returned result. Although all statements of the method have an influence on the object to be asserted, the test case will not detect (subtle) faults in the code because it only asserts that the object is not null.

```
1 public static Rectangle createRectangle(double area,
    ↪ double aspectRatio) {
2     if (area < 0 || aspectRatio <= 0) {
3         throw new IllegalArgumentException();
4     }
5     double width = determineWidth(area, aspectRatio);
6     double height = determineHeight(area, aspectRatio);
7     return new Rectangle(width, height);
8 }
9
10 @Test
11 public void testCreateRectangle() {
12     assertNotNull(createRectangle(20.0, 1.5));
13 }
```

Listing 3.1: Test case with a not very effective assertion.

Another example of a test case with weak assertions is one that invokes a method returning a list and only checks whether the list as such is not empty or matches a specified length, without performing checks on the list elements. Even `assertEquals` assertions may not always validate an object in a perfect way. This is because such an assertion uses the `equals` method of the particular object's class to determine whether two objects are considered equal. An `equals` method does not necessarily compare the whole state of an object but may consider only some of the fields in the comparison. Hence, deviations in fields that are not checked remain undetected.

The validity of mutation testing does not rely on the effectiveness of assertion statements.

### 3.3 Mutation Testing

Mutation testing takes fault detection capabilities into account. Therefore, it is not surprising that many studies demonstrated that mutation testing is a good indicator of test effectiveness. This has, for example, been shown in the empirical studies of [Andrews \*et al.\* \(2005\)](#), [Chekam \*et al.\* \(2017\)](#), [Daran and Thévenod-Fosse \(1996\)](#), [Frankl \*et al.\* \(1997\)](#), [Gopinath \*et al.\* \(2014a\)](#), [Hutchins \*et al.\* \(1994\)](#), [Just \*et al.\* \(2014a\)](#), [Li \*et al.\* \(2009\)](#), [Mathur and Wong \(1994\)](#), [Offutt \*et al.\* \(1996b\)](#), [Papadakis \*et al.\* \(2018\)](#), and [Ramler \*et al.\* \(2017\)](#).

[Just \*et al.\* \(2014a\)](#) also confirmed the validity of mutation testing because they found that mutants are valid substitutes for real faults. Consistent with this observation is the empirical evidence provided by [Andrews \*et al.\* \(2006\)](#) and [Daran and Thévenod-Fosse \(1996\)](#) showing that the mutation score correlates with actual failure rates.

In the following, we briefly summarize related work addressing mutation testing's computational complexity and the problem of equivalent mutants, describe the use of mutation testing in practice, and present related work on pseudo-testedness.

#### 3.3.1 Addressing the Computational Complexity

Below, we summarize work on effort reductions, incremental mutation testing, and models to predict mutation testing verdicts.

##### 3.3.1.1 Effort Reductions

Researchers have suggested several approaches to reduce the computational complexity of a mutation analysis. [Offutt and Untch \(2001\)](#) classified these approaches as *do fewer*, *do smarter*, and *do faster*. *Do fewer* approaches



comprise the use of a smaller, representative set of mutation operators (Namin *et al.*, 2008; Offutt *et al.*, 1993; Offutt *et al.*, 1996a), sampling of mutants (Acree Jr., 1980), mutants clustering (Ji *et al.*, 2009), and higher order mutation, in which multiple mutation operators are applied at once (Jia and Harman, 2008). The most prominent *do smarter* approach is weak mutation, in which a mutant is immediately evaluated after its execution point instead of checking it at the end of a test execution (Howden, 1982; Jia and Harman, 2011). *Do faster* approaches comprise further run-time optimization techniques to speed up the generation and execution of mutants (e.g., byte-code mutants (Ma *et al.*, 2005; Schuler *et al.*, 2009), aspect-oriented mutation (Bogacki and Walter, 2006), or parallel mutation testing (Fleyshgakker and Weiss, 1994)).

Jia and Harman (2011) later grouped these approaches into the *reduction of the generated mutants* (“do fewer”) and the *reduction of the execution costs* (combining “do faster” and “do smarter”).

### 3.3.1.2 Incremental Mutation Testing

Zhang *et al.* (2012) proposed “regression mutation testing” to reduce efforts by incrementally updating mutation testing verdicts. Their approach compares control-flow graphs of two program versions to identify edges that may lead to different test behavior, and decides based on the outcome which mutants need to be re-assessed. They evaluated their approach on six study objects with five revisions each, and the revisions contained up to eight changed files. The reported reduction rate is between 0.08% and 100%. A similar concept was presented by Cachia *et al.* (2013). Their approach works by isolating test cases that invoke changed methods from the remaining test cases. However, when identifying these test cases, the approach does not consider interactions between methods. Their preliminary evaluation on the project APACHE COMMONS CLI LIBRARY with 5 k lines of code (LOC) showed that speed improvements between 88% and 91% can be achieved. Bajada *et al.* (2015) extended this approach by considering the control flow to more precisely identify the effects of changes between two versions of a

software. Their study on two small study objects revealed that numerous relevant mutants were generated that were skipped by the basic approach. They found that the costs for the extended approach are considerably higher than for the basic approach, but concluded that it remains advantageous compared to full mutation testing when the increment is small. They did not assess how many mutants that are affected by changes were erroneously skipped.

The mutation testing tool *PIT* provides an experimental feature to perform incremental mutation testing on Java applications (see Chapter 2.4.3). Its performance has not been empirically validated yet.

### 3.3.1.3 Predicting Mutation Testing Results

Prediction models have been employed in a few mutation testing studies. [Namin \*et al.\* \(2008\)](#) used linear models and [Jalbert and Bradbury \(2012\)](#) applied machine-learning models to predict the overall mutation score without executing a mutation analysis. Both did not perform predictions on individual mutants.

[Strug and Strug \(2012\)](#) and [Strug and Strug \(2018\)](#) used machine learning to reduce the number of mutants to be executed. They calculated the structural similarity of mutants and predicted based on results of similar mutants whether a given test would detect a mutant or not. However, their approach still requires a mutation analysis of a subset of mutants.

The most related work to our machine-learning model for pseudo-tested methods is from [Zhang \*et al.\* \(2018\)](#) who predicted the mutation testing verdict of individual mutants with promising precision and recall values. However, it is difficult to judge their results because they also included mutants that are not covered by any test case. These mutants cannot be killed and are therefore trivial to predict based on line coverage. In Chapter 6, in contrast to their work, we predict the mutation testing verdict of a method and not of a single mutant, exclude methods that cannot be killed since they are not covered, and include the proposed minimal stack distance measure in the prediction model.

### 3.3.2 Addressing Equivalent Mutants

[Budd and Angluin \(1982\)](#) proved that the equivalence of two programs is generally undecidable. Consequently, it is not possible to fully automatically detect equivalent mutants. Since equivalent mutants distort the results and the manual analysis of mutants is labor-intensive, researchers have suggested a number of heuristics to uncover them.

[Baldwin and Sayward \(1979\)](#) proposed an approach, which uses compiler optimization techniques to identify equivalent mutants. The idea behind that is that some equivalent mutants exhibit the same code after applying optimization or de-optimization rules to them. [Offutt and Pan \(1996\)](#) and [Offutt and Pan \(1997\)](#) devised a constraint-solving approach, which analyzes a mutant's path conditions. Other researchers used program slicing to assist the detection of equivalent mutants ([Harman et al., 2001](#); [Hierons et al., 1999](#); [Voas and McGraw, 1997](#)). [Grün et al. \(2009\)](#) measured the impact of mutants on program execution and suggested that mutants with higher impact exhibit a greater “killability” and are less likely equivalent. More recently, [Papadakis et al. \(2015\)](#) incorporated their “Trivial Compiler Equivalence (TCE)” approach into the *MiLu* mutation testing tool. They claimed that it is the first tool that supports a fully automated detection of (some of the) equivalent mutants. [Kintis et al. \(2017\)](#) conducted an empirical study on TCE and reported that this technique can uncover approximately half of all equivalent mutants in Java.

### 3.3.3 Use In Practice

Below, we briefly summarize research that focuses on the practical applicability of mutation testing.

[Nica et al. \(2011\)](#) analyzed the performance of three mutation testing tools (*MuJava*, *Jumble*, and *Javalanche*) and concluded that mutation testing is too slow to be used in real-world software projects. Another comparison of five mutation testing tools, including *PIT*, was conducted by [Delahaye and Du Bousquet \(2013\)](#). They showed that all but one tool were able to

complete the analysis for every study object. However, they pointed out that the tools need a lot of improvements to be ready for the use in real-world projects. The mutation testing tool *PIT* was also assessed by [Klischies and Fögen \(2016\)](#). They stated that the analysis is “fast enough for small to medium sized projects such as utility libraries,” but expressed strong concerns with regard to applicability to larger projects due to the required run-time.

The maturity of mutation testing and different strategies to reduce the run-time overhead were investigated by [Možucha and Rossi \(2016\)](#). They suggested that selective mutation operators, second order mutation, and multi-threading can help increase the applicability. According to them, the obtained results are influenced by the use of these strategies such that a trade-off between speed and result quality needs to be found. They concluded that “mutation testing is mature enough to be more widely adopted.”

[Ramler et al. \(2017\)](#) reported the results of a case study on mutation testing conducted with an engineering company developing safety-critical systems. The investigated system was implemented in C and comprised 60,000 LOC. The mutation analysis generated 75,043 mutants out of which 63.8% were killed by unit tests. The engineers investigated a sample of 200 live mutants and found at least 24 of them to be equivalent and at least 12 to be duplicated. According to the study, the analysis provided valuable hints on how to improve the test suite, and the engineers identified two new faults in the code when improving the tests. However, the mutation analysis took 4,000 hours computing time. Another case study on a complex real-world project was conducted by [Ahmed et al. \(2017\)](#). They applied mutation testing to a module of the Linux kernel and found mutation testing to be a useful tool because it helped them uncover two faults, although the module is “well tested and heavily used.” The mutation analysis took 3,499 hours, whereof 30 minutes each were needed to compile a mutated version. They discarded mutants that did not compile or were considered as not interesting and thereby reduced the number of mutants to be analyzed from 3,169 to 380. 10% of the mutants were found to be equivalent and 20% to be duplicated.

More recently, [Petrović and Ivanković \(2018\)](#) presented how mutation testing is employed at Google. The development process at Google specifies that code changes go through a review process before they are merged into the main branch of the repository. When a code diff is sent for review, an incremental coverage analysis is conducted to identify the set of covered lines that concern changes. Then, the mutation analysis is triggered. It first filters out lines with statements that are considered as not interesting (e.g., logging statements). It then generates at most one mutant for each remaining line, in which the mutation operator is randomly selected from the set of applicable operators for the given line. Finally, unit tests are executed against the mutants, and surviving mutants are presented to the developers in the code review tool. Developers can give feedback on the mutants to improve the exclusion of uninteresting mutants. [Ivanković et al. \(2018\)](#) revisited this case study by [Petrović and Ivanković](#) and identified challenges for a widespread adoption of mutation testing. First, they found that “unproductive mutants” exist, which are either not relevant, equivalent, or redundant. Developers do not want to spend time on them. Second, they pointed out that a developer’s workflow is commit-centered such that mutation testing needs to be integrated into that workflow. Third, more mutants are generated than developers can analyze in a reasonable amount of time. The challenge is to present the most relevant mutants to developers.

#### 3.3.4 Pseudo-Testedness

[Vera-Pérez et al. \(2017\)](#) also investigated pseudo-testedness and replicated our study published in [Niedermayr et al. \(2016\)](#). Pseudo-tested methods equally exist in all their study objects, even in ones with high coverage. Thereby, they confirmed our observation that pseudo-tested methods are common.

They further studied the mutation score of pseudo-tested methods that is achieved when conducting an analysis with traditional mutation operators. They found that the mutation score of these methods is not always 0%, but it is substantially lower than the one of the remaining covered methods.

Nonetheless, pseudo-tested methods exist that reach high mutation scores due to “trivial exception-raising mutants.” Their study objects contain 63 pseudo-tested methods with 100% mutation score; approximately 54% of these have only one or two mutants, but one method has 69 mutants, which are all killed by thrown exceptions.

They manually analyzed pseudo-tested methods with void and boolean return type, and identified misplaced or missing oracles, too weak oracles, and missing test input as issues causing these methods to be pseudo-tested.

Finally, [Vera-Pérez \*et al.\*](#) conducted developer interviews to get more insights into the practitioner’s view on pseudo-tested methods. According to the interviews, developers considered flaws identified by these methods as easy to understand and confirmed their relevance. The developers accepted pull requests that were provided during this study to improve the testing state of pseudo-tested methods. However, when they only received information about the problem without pull request, they did not always improve the test suite due to limited resources and other pending tasks. According to the developers, pseudo-tested methods are especially relevant and deserve additional testing efforts when methods concern the core functionality of a project or are widely used within a project. By contrast, methods that are automatically generated or concern debugging or logging functionalities belong to the ones considered as less relevant.

[Delplanque \*et al.\* \(2019\)](#) coined the term “rotten green tests,” which are passing test cases that contain assertions, but at least one assertion is never executed. Such tests give developers a false sense of confidence because they contribute coverage without performing all intended checks. [Delplanque \*et al.\*](#) believe that rotten green tests are a possible cause for pseudo-tested methods.

### 3.4 Defect Prediction

The idea behind defect prediction is to predict code areas that are especially fault-prone ([Catal and Diri, 2009](#); [Hall \*et al.\*, 2012](#)). Defect prediction allows

software engineers and testers to focus quality-assurance efforts on these areas and thereby supports a more efficient resource allocation (Menzies *et al.*, 2007; Weyuker and Ostrand, 2008). The prediction models can be built using code metrics (D'Ambros *et al.*, 2012; Menzies *et al.*, 2007; Nagappan *et al.*, 2006; Zimmermann *et al.*, 2007), change metrics (Hassan, 2009; Kim *et al.*, 2007; Nagappan and Ball, 2005), or a variety of further metrics (such as code ownership (Bird *et al.*, 2011; Rahman and Devanbu, 2011), developer interactions (Lee *et al.*, 2011; Meneely *et al.*, 2008), dependencies to binaries (Zimmermann and Nagappan, 2008), mutants (Bowes *et al.*, 2016), code smells (Palomba *et al.*, 2016), and others).

Defect prediction is usually performed at the component, package or file level (Bacchelli *et al.*, 2010; Nagappan and Ball, 2005; Nagappan *et al.*, 2006; Scanniello *et al.*, 2013). More fine-grained prediction models have been proposed to narrow down the scope for quality-assurance activities. Kim *et al.* (2008) presented a model to classify code changes. Hata *et al.* (2012) applied defect prediction at the method level and showed that fine-grained prediction outperforms coarse-grained prediction at the file or package level if efforts to find the faults are considered. Giger *et al.* (2012) also investigated predictions at the method level and concluded that a random forest model operating on change metrics can achieve good performance. More recently, Pascarella *et al.* (2018) replicated this study and confirmed the results. However, they reported that a more realistic inter-release evaluation of the models shows a dramatic drop in performance with results close to that of a random classifier, and concluded that method-level bug prediction is still an open challenge. It is considered difficult to achieve sufficiently good data quality at the method level (Hata *et al.*, 2012; Shippey *et al.*, 2016); publicly available datasets have been provided in Giger *et al.* (2012), Just *et al.* (2014b), and Shippey *et al.* (2016).

Cross-project defect prediction predicts faults in projects for which no historical data exists by using models trained on data of other projects (Xia *et al.*, 2016; Zimmermann *et al.*, 2009). He *et al.* (2012) investigated the usability of cross-project defect prediction. They reported that this type of prediction works only in few cases and requires careful selection of train-

ing data. [Zimmermann et al. \(2009\)](#) also provided empirical evidence that cross-project prediction is a serious problem. They stated that projects in the same domain cannot be used to build accurate prediction models without quantifying, understanding, and evaluating process, data and domain. Similar findings were obtained by [Turhan et al. \(2009\)](#), who investigated the use of cross-company data for building prediction models. They found that models using cross-company data can only be “useful in extreme cases such as mission-critical projects, where the cost of false alarms can be afforded,” and suggested using within-company data if available. While some recent studies reported advances in cross-project defect prediction ([Xia et al., 2016](#); [Xu et al., 2018](#); [Zhang et al., 2016](#)), it is still considered as a challenging task.

In Chapter 7, we identify methods that we consider as “too trivial to test”. While a lot of research has been conducted to predict fault-prone artifacts, our work is the first that explicitly focuses on methods with a low fault risk.

### 3.5 Test Case Selection and Prioritization

Regression testing is an activity to ensure that conducted changes do not harm the existing behavior of a software ([Yoo and Harman, 2012](#)). A common procedure for regression testing is to execute all existing test cases of a test suite (*retest-all*). However, as software evolves, the number of automated test cases increases over time such that the execution time of the whole test suite rises. Consequently, it is often not feasible or too costly to execute the whole test suite after each change. It is sometimes even not possible to execute the all tests before an upcoming release. Therefore, various approaches have been developed and studied to reduce the efforts for regression testing ([Graves et al., 2001](#)). The main approaches are test suite minimization, test case selection, and test case prioritization. Latter two are related to our work in Chapter 8.

*Test case selection* techniques aim to reduce regression testing efforts by deriving an appropriate subset of existing test cases ([Rothermel and Harrold,](#)



1996); that is, these techniques determine which minimal set of test cases needs to be re-executed to check the behavior of the software after changes (Engström *et al.*, 2010). Unlike test suite minimization, test case selection techniques do not permanently remove tests from a test suite (Rothermel *et al.*, 1998). *Test case prioritization* techniques schedule test cases in such an order that test cases considered more important (in respect to a given goal) are executed earlier in the test suite execution (Rothermel *et al.*, 2001). The prioritization goals can be to detect faults faster in the testing phase (Catal and Mishra, 2013; Rothermel *et al.*, 1999), to increase overall code coverage as soon as possible (Elbaum *et al.*, 2002), or to cover important features earlier (Rothermel *et al.*, 2001).

The vast majority of existing research literature deals with either the selection or the prioritization of test cases, but does not combine both (Silva *et al.*, 2016). To the best of our knowledge, the first approach that combines selection and prioritization was presented by Wong *et al.* (1997). It performs a test selection based on modified code followed by a prioritization of the remaining test cases, which considers a test case's increasing cost per additional coverage. In contrast to our work, they did not include test effectiveness information.

Singh *et al.* (2010), Suri and Singhal (2011), and Walcott *et al.* (2006) presented test prioritization approaches that additionally incorporate a time constraint for the test execution. Thereby, their approaches implicitly also conduct a test selection by excluding lower-priority test cases that do not fit into the time frame. However, the selection does not take code changes (since the previous test execution) into account.

Silva *et al.* (2016) combined selection and prioritization in a hybrid approach, which is conducted in five stages. Their approach first maps features to software classes to determine their relevance. Next, it calculates the criticality of classes based on coupling, complexity, and relevance. Afterwards, it computes the criticality of tests by taking into account what classes they cover. It then identifies test cases that cover at least one changed class and applies an Ant Colony Optimization algorithm to select the set of the most critical tests that can be executed within available time. Finally, tests are

ordered based on their criticality. In contrast to their work, we do not take the relevance of features into account, operate on method coverage instead of class coverage, use a different prioritization criterion, and additionally consider test effectiveness information.

More recently, [Spieker et al. \(2017\)](#) developed a history-based test case selection and prioritization approach integrated into continuous integration. It prioritizes test cases based on their meta-data, last execution time, and previous results. Then, it selects the most important test cases. The approach uses reinforcement learning to learn from experiences of previous test executions; that is, it “can progressively improve its efficiency from observations of the effects its actions have.” Their work does not use coverage and change information for test case selection.

Several approaches that perform test case prioritization by considering a test’s fault exposing potential have been proposed in [Elbaum et al. \(2002\)](#), [Farooq and Nadeem \(2017\)](#), [Gonzalez-Hernandez et al. \(2018\)](#), [Lou et al. \(2015\)](#), [Rothermel et al. \(1999\)](#), [Rothermel et al. \(2001\)](#), and [Shin et al. \(2019\)](#). In contrast to their work, we conduct a change-based test case selection before the prioritization, and use test effectiveness information at another level. While they apply mutation testing to compute the overall effectiveness score of a test case, we compute pseudo-testedness information of a test-case method pair.

### 3.6 Research Gap

Mutation testing has rarely been adopted as a test adequacy criterion in practice. Reasons for this include its run-time complexity, the problem of equivalent mutants, and the lack of integration into development workflows ([Madeyski et al., 2014](#)).

In this dissertation, we aimed to make mutation testing better applicable and more actionable. To achieve this, we introduced the concept of pseudo-testedness and devised a mutation operator to detect pseudo-tested methods. The mutation operator works at the method level, controls the computational

complexity by limiting the number of mutants, and prevents equivalent mutants. Unlike mutants generated by more fine-grained mutation operators, pseudo-tested methods are easy for developers to understand. Consequently, these methods come as analysis result to the fore. They are, unlike the mutation score, actionable such that developers can address them to improve the test suite. This is what DeMillo *et al.* (1978) initially had in mind in the early days of mutation testing, when stating that mutants (and not the score) matter because they advise designing better test cases.

To make analysis results even more relevant and further speed up the computation, we also studied in this dissertation which methods are not prone to faults, and therefore too trivial to test, such that they can be excluded from a mutation analysis.



# CONCEPT OF PSEUDO-TESTED METHODS

*This chapter introduces the concept of pseudo-testedness. It describes the idea as well as the advantages of this concept and explains its realization as a mutation operator.*

## 4.1 Motivation

Mutation testing is a technique for measuring the fault detection capabilities of a test suite by introducing faults into the application code and evaluating whether test cases can uncover them. While mutation testing is well suited to providing insights into the testing state, it suffers from two major problems.

First, mutation testing is computationally complex. In the worst case, all test cases need to be executed for each mutant to determine whether a mutant can be killed or not. Consequently, the asymptotic complexity of mutation testing for a project is  $\mathcal{O}(|M| \cdot |T|)$ , with  $|M|$  being the number of methods (or, in non-object-oriented languages, functions) and  $|T|$  the number of test cases. Accordingly, the analysis effort increases polynomially with an increasing number of methods and test cases. Moreover, a significant number of mutants may be created for each method, depending on a method's length and the used constructs in its body. [Acree \*et al.\* \(1979\)](#) suggested that the number of mutants is in the order of the square of the number of lines in the program. For example, *Mothra*, an early mutation testing tool for the FORTRAN language, “generates 970 mutants for a particular 27-line program” ([King and Offutt, 1991](#)). Regarding the asymptotic complexity, the (maximum) number of mutants per methods can be assumed to be constant because it is independent from the size of a project.

In practice, not all test cases cover all methods. Therefore, the actual effort required to examine a mutant of a method is usually lower than that required to execute the entire test suite. Optimizations allow an early abort of the analysis of a mutant after the first test case that kills the mutant is found. Nonetheless, mutation testing is an order of magnitudes more expensive than a regular execution of a test suite or a computation of the code coverage. Efforts are especially high in large projects that contain long-running integration or system tests.

In large industry projects with comprehensive automated test suites, even the regular execution of the whole test suite may take a couple of weeks so that the test suite's duration may already constitute a problem (see [Section 8.1](#)). In such projects, mutation testing is generally not applicable due

Table 4.1: Duration of a mutation analysis using PIT (in hours) compared to a regular test suite execution.

Project	LOC	Test suite execution	Mutation analysis	Slow-down factor
BIOJAVA	240.6 k	00:27:45	34:57:00	75.6
BITCOINJ	59.1 k	00:01:26	02:44:00	114.4
JFREECHART	222.8 k	00:00:13	00:25:37	118.2
PDFBOX	227.6 k	00:01:25	04:30:00	190.6

to the amount of effort required.

A lot of research effort has been invested in addressing the computational complexity of mutation testing (see Section 3.3.1). PIT is one of the most advanced mutation testing tools for Java (Delahaye and Du Bousquet, 2013; Gopinath *et al.*, 2017) and is equipped with several optimization techniques (Coles *et al.*, 2016). For many projects, however, mutation testing is not fast enough to be applicable. Table 4.1 demonstrates the cost of mutation testing for four Java projects by presenting the duration of the test suite execution, the duration of the mutation analysis using PIT with traditional mutation operators, and the resulting slow-down factor. In these projects, mutation testing takes 75.6–190.6 times as long as the regular test suite execution.

A second drawback of mutation testing is that the analysis may generate and evaluate equivalent mutants. Equivalent mutants are semantically equal to the original code, which means that they cannot be killed (see Section 2.4.1). It is not possible to automatically detect and filter such mutants because the equivalence of two functions is generally not decidable (Budd and Angluin, 1982). The execution of tests for equivalent mutants causes needless efforts during the mutation analysis. Even more importantly, they distort the results and threaten the developer’s acceptance of mutation analysis results. Developers need to manually inspect the mutants that were not killed to determine whether they are equivalent (and therefore irrelevant) or whether tests need to be improved. Schuler and Zeller (2013b) conducted an experiment and found that the manual classification of a mutant by a

developer takes 15 minutes on average. Grün *et al.* (2009) found equivalent mutants to be surprisingly common. Hence, equivalent mutants are an obstacle to the use of mutation testing in industry.

In sum, despite several optimization techniques, mutation testing is still computationally complex and carries the problem of equivalent mutants. In industry, mutation testing lacks adoption (Ivanković *et al.*, 2018; Jia and Harman, 2011; Madeyski *et al.*, 2014). Instead, code coverage is used broadly as a measure for test suites (Hemmati, 2015; Smith and Williams, 2008; Yang *et al.*, 2009), although, it is known and considered as a problem that code coverage does not measure test effectiveness.

Parts of the content of this chapter have been published in Niedermayr *et al.* (2016). A preliminary version of the mutation operator has been presented in Niedermayr (2013).

## 4.2 Idea

We propose a light-weight mutation testing approach that operates at the method level to address both downsides of mutation testing. This approach is designed to identify methods that are covered by test cases, but where none of the test cases can detect whether the method's whole logic is removed. That is, although these methods are invoked by test cases, they are actually not tested at all. We call such methods *pseudo-tested methods*. The idea behind pseudo-testedness is that if no single test case can detect such an invasive transformation, no test case will generally be able to detect subtler faults.

More precisely, we define a method as *covered* if it is partially or fully executed by at least one test case. We define a method as *pseudo-tested* if it is covered, not empty, and no test case fails on any mutant in which all original statements were removed. By contrast, we consider a method as *tested* (with respect to this mutation operator) if at least one test case fails when the whole logic of the method is removed.

For methods that declare a return type other than void, it is necessary



to add a return statement with a value of an appropriate type to keep the method's code compliant with the signature. To avoid a method from turning into an equivalent mutant when its original logic solely returns a value, two mutants with different return values may be generated. In this case, a method is pseudo-tested if no test case fails on any of both mutants. Consequently, the number of mutants per method is limited and does not exceed two.

A mutation analysis with a mutation operator for pseudo-tested methods has the following advantages over an analysis with traditional, more fine-grained mutation operators.

- (1) An analysis using this mutation operator is *faster*. It generates at most two mutants per covered method, which keeps the overall number of mutants minimal. Furthermore, due to the invasive transformation, a mutant is easier to kill with the consequence that more test cases will detect the mutation. Since the analysis of a mutant can be stopped as soon as the first test case kills the mutant, fewer tests will need to be executed per mutant.
- (2) This mutation operator produces *hardly any equivalent mutants*. A more invasive transformation is generally less likely to generate an equivalent mutant than a small, subtle change. [Schuler and Zeller \(2013b\)](#) provided empirical evidence that mutation operators that manipulate the control flow have a high impact on the behavior of a program and produce fewer equivalent mutants than operators that change data. Our approach also undertakes further actions to reduce the risk of equivalent mutants. First, certain methods (*e.g.*, hashCode methods or empty methods, which may be present to satisfy an implemented interface) are excluded from the analysis. Second, for methods that return a primitive value, two mutants that differ in the return value are generated. A method is only considered pseudo-tested if both mutants cannot be killed. Since both mutants return a different value, and therefore differ semantically from each other, it is impossible for both to be equivalent to the original method.

While these actions do not guarantee that absolutely no equivalent mutants are generated,<sup>1</sup> they strongly reduce their probability. This is a significant advantage because [Schuler and Zeller \(2013b\)](#) found in their study using the mutation testing tool *Javalanche* ([Schuler and Zeller, 2009](#)) that approximately 45% of the undetected mutants in seven Java projects were equivalent. In a more recent study by [Carvalho et al. \(2018\)](#) on configurable industrial systems written in C, nearly 40% of the mutants were found to be equivalent.

Besides equivalent mutants, duplicated mutants, which are equivalent between them but not with the original version ([Papadakis et al., 2015](#)), cannot occur with our mutation operator.

- (3) The mutation operator is *easy* for developers to comprehend and the generated mutants are easy to interpret. It is simple for a human to imagine what the consequences of the mutation are. One does not need to understand the control flow of a method since the mutation transforms the whole method body. This is not the case for a fine-grained mutation operator, which, for example, negates the expression of a conditional statement in a long, nested method. [Vera-Pérez et al. \(2017\)](#) confirmed that problems identified by pseudo-tested methods are easy for developers to understand.
- (4) The mutation operator is *actionable*. [Hilton et al. \(2018\)](#) stated that “developers should consider using more detailed metrics than just the ratio of statements covered to measure their code’s testedness.” An analysis with our operator does not just produce a score, but it does provide a list of methods that are pseudo-tested. Consequently, it allows developers to improve their test suite by addressing these methods. To do so, developers can design new test cases to address pseudo-tested methods or enhance existing test cases by adding further assertions.

---

<sup>1</sup> An example of a method that will still turn into an equivalent mutant would be one that returns no value and solely invokes empty methods; that is, it only delegates to methods without logic. Heuristics could be used to detect and exclude such methods. However, it is not possible to detect all cases using static code analysis because Java employs dynamic method dispatch ([Miglani, 2017](#)).

The concept of pseudo-testedness is a light-weight way to identify methods that are insufficiently tested such that tests cannot uncover semantic faults in them. However, tests may still be able to detect faults in pseudo-tested methods that result in run-time errors. Even the execution of a method without any assertions may uncover, for example, null-pointer de-reference exceptions or arithmetical exceptions caused by divisions by zero. In other words, while tests cannot detect incorrect computations in pseudo-tested methods, they may still be able to ensure the absence of thrown exceptions.

The fact that a method is not pseudo-tested does not imply that it is effectively tested. Traditional mutation operators are more fine-grained, operate at the statement level, and may reveal living mutants in statements of methods that are not pseudo-tested. Therefore, we intuitively consider the proportion of methods that is not pseudo-tested as the upper bound of the proportion of methods that is effectively tested.

In sum, the presented light-weight mutation operator identifies covered methods that are actually not tested at all (except in terms of thrown exceptions). Hence, its outcome is more valid in terms of fault detection capabilities than code coverage. At the same time, the approach is more efficient than a mutation analysis with traditional, more fine-grained mutation operators, and addresses the problem of equivalent mutants.

### 4.3 Definition of the Mutation Operator

The mutation operator to uncover pseudo-tested methods works at the method level and removes the whole content from a method's body. An appropriate return statement is added to the mutated method if it returns a value; therefore, the mutation operator design depends on the return type of the method.

*Methods with void as return type:* All statements of the method body are removed so that the original method logic is no longer executed. No further actions are necessary for methods of this type. To avoid equivalent mutants, methods that are already empty are not mutated.

Table 4.2: 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"

*Methods with a primitive return type<sup>1</sup> or string return:* As with void methods, the mutation operator removes the original code from the method. A return statement is then added to the method, which returns a constant value that satisfies the specified return type. The values per type are specified in Table 4.2. Two values are provided per type because two different mutants are generated for methods of this type. A method is only considered pseudo-tested if neither of the mutants can be killed by any test case. The use of two mutants prevents equivalent mutants because it is not possible that both mutants are equivalent to the original method.

*Methods with complex return types:* The mutation operator removes the code from the method body and adds a return statement, which invokes a factory to create a suitable instance (object) as return value. The factory is already invoked once for a method during the mutation process to determine whether it is able to provide an instance for a given type. If the factory cannot create an instance that satisfies the method's return type, the method will not be mutated. Listing 4.1 presents an example of a mutated method.

A factory takes the qualified name of a method's declared return type as input. It then creates an instance that satisfies the given type or throws an exception to signalize that it does not support the type.

---

<sup>1</sup> For Java, we also count methods that return a class of a primitive wrapper type to this category.

```

1 public static FileTreeWalker fromJarFile(String pathToJarFile)
   ↳ throws IOException {
2   JarFileContent jarFileContent = new JarFileContent(new JarFile(
   ↳ pathToJarFile));
3   jarFileContent.initEntries();
4   return jarFileContent.getVisitor();
5   return TestAnalyzerEnvironment.getCurrentFactory().createInstance(
   ↳ "java.nio.file.FileTreeWalker");
6 }

```

Listing 4.1: Example of a mutant that returns a non-primitive value.

A factory should satisfy two requirements. First, for every invocation with the same type, a factory should create a semantically equal instance. This is necessary for the mutation operator and the mutant to behave deterministically. For example, each instance of the `Date` class needs to be created with the same time stamp and each instance of a random value generator needs to be created with the same seed value. Second, a factory should create a new instance at every invocation instead of reusing instances. This avoids side-effects and satisfies the first requirement when the instance's state is changed during the test execution. However, it may not be possible to create a new instance for each type (e.g., for types that follow the singleton pattern).

A factory may use different techniques to create instances and can delegate to other factories. This may work as follows. First, the factory checks whether the requested type is a common Java class or interface for which the factory provides dedicated logic to create instances. Thus, instances can be generated, for example, for the types `Runnable`, `Optional`, `Iterable`, `Date`, and others. Next, if a factory does not explicitly support a type,<sup>1</sup> it can use reflection to create instances of classes that exhibit a public constructor without parameters. With the same technique, it can further create instances of classes that exhibit a public constructor that only takes primitive values,

<sup>1</sup> Certain types should be handled manually or be excluded because they may cause undesired effects when used in mutants during the test execution. An example is the `File` class, which could lead to a file system erasure in the worst case.

string values, arrays, or collections as input; the factory will use the values of mutant 1 from Table 4.2, or zero-length arrays, or empty collections as arguments for the parameters. Furthermore, the factory can instantiate single- and multi-dimensional arrays of arbitrary types using reflection; the created arrays will be of length 0. Instances of enum values can be provided by picking the first declared enum constant. If it has not been possible to create an instance so far, further code tailored to the system under test can be developed in order to provide instances of project-specific types for which the previous steps have not been successful.

Since only one mutant is created per method with a complex return type, equivalent mutants may occur in seldom cases when a method solely creates and returns an instance of the return type.

## Excluded Methods

For Java code, the following methods should always be excluded from a mutation analysis with this mutation operator.

- *Empty methods* contain no statements and are of no interest because they contain no logic that could be tested. More importantly, they would result in an equivalent mutant.
- Implementations of Java's *hashCode method*<sup>1</sup> cannot be tested with mutation testing in a meaningful way. This is because a *hashCode* method still fulfills its specification if another computation formula is used, even if it includes fewer fields in its computations. This implies that a method that always returns the same constant value is in line with the specification. The specification will only be violated if additional fields of the enclosing class are included in the computations. The presented mutation operator does not introduce any additional field accesses (nor is that common for traditional mutation operators). To validate *hashCode* methods, other techniques should be used, such as, the *EqualsVerifier*<sup>2</sup>.

---

<sup>1</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode>

<sup>2</sup> <https://jqno.nl/equalsverifier>

- Special *methods introduced by the compiler* should not be tested. They are not present in the source code such that developers are not aware of these methods. They are automatically inserted into the byte code during compilation. Examples of such methods are implicit constructors, an enum's `valueOf` and `values` methods, and synthetic methods that the JVM needs for handling nested classes and type erasure (Gosling *et al.*, 2015).
- It is necessary to exclude *methods* with return types for which *no appropriate value* can be generated. This case is only relevant for methods that return a non-primitive value. The used factory may be unable to provide an instance of a type (e.g., due to restrictive access modifiers or when no class implementing a certain interface is available on the classpath), may not address a certain return type for another reason, or may fail to create an instance at run-time without causing an exception. If no appropriate value can be generated, it will not be possible to conduct the mutation because no compilable method can be generated (unless `null` will be used as an alternative return value).

## 4.4 Implementation

In this section, we describe how a mutation analysis with the presented mutation operator can be performed for Java programs. We implemented this approach in our *TestAnalyzer*<sup>1</sup> tool.

The mutation approach consists of four steps and is depicted in Figure 4.1. 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 covered method and can run concurrently.

- (1) *Instrumentation*: In the first step, we need to instrument the code. We instrument methods of the application code by inserting a statement

---

<sup>1</sup> <https://github.com/cqse/test-analyzer>

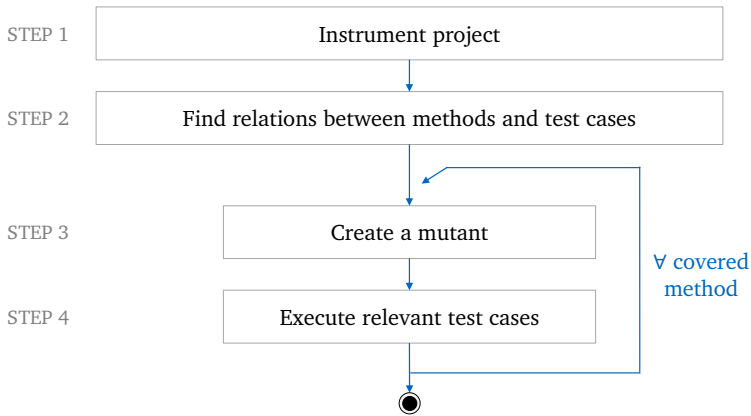


Figure 4.1: Overview of the mutation process. The first two steps of the mutation analysis are executed once, the latter ones are executed for each considered method.

at the beginning of each method, which invokes a recorder class with the method identifier. The method identifier is unique and composed of the qualified name of the enclosing class, the method name, and the parameter types.

Furthermore, we instrument the test code. We insert a statement at the beginning and at each exit point of each test case. This allows us to determine when the actual test execution takes place so that we can differentiate between the actual test execution and set-up as well as tear-down procedures happening before or after the test (e.g., methods annotated with JUnit’s @Before or @After annotation).

- (2) *Computation of test-case method relationships*: In the next step, we determine the relationships between test cases and methods. To do so, we execute each test case once on the instrumented code to find out which methods it covers during the actual execution. We do not consider methods that are invoked only during a test’s set-up or tear-down procedures because they are not in the intended scope of a test.



After having executed all test cases, we know for each test case which methods it covers and can compute for each method by which test cases it is covered. Test cases that fail at this point are excluded from further analysis.

- (3) *Creation of mutants*: In the next step, non-empty methods that are covered by at least one test case are mutated one after another. To do so, the mutation operator described in Section 4.3 is applied to the byte code. The operator removes the whole logic from a method's body and adds an appropriate return statement if needed. Methods for which no suitable return value can be generated will be skipped.
- (4) *Test executions against mutants*: For each mutant, a new JVM process is created and all test cases that cover the method are executed against the mutated code. A mutant may be killed by a test failure or a time-out caused by an endless loop resulting from the mutation. The determination of the time-out for a mutant takes the number and duration of the mutant's covering test cases into account.

The outcome of this step shows which test cases fail and which ones still succeed after the mutation of a method. The analysis of a mutant is not aborted after it is killed by the first test case so that a full mutation matrix can be computed. Such a matrix allows determining whether a method is pseudo-tested by a particular test case.

#### 4.4.1 Configuration Options

The execution of the mutation analysis is fully automatic. The analysis takes a configuration file as input, which specifies:

- the code paths pointing to the application code to be mutated. The paths can either be jar files or directories containing the compiled class files.
- the code paths pointing to the test code containing the test cases. The paths can either be jar files or directories containing the compiled class files.

- the class path, which references additional libraries that are needed to execute the system under test.
- the test runner that shall be used. This can be the JUnit or TestNG runner provided by the *TestAnalyzer* or another externally provided test runner tailored to a certain project. The test runner is in charge of identifying test classes, identifying test cases in test classes, and triggering the execution of a single, atomic test case. For example, the test runner for JUnit identifies test classes by searching for classes that inherit from `TestCase` (JUnit 3) or contain methods annotated with `@Test` (JUnit 4).
- the return value generators to be used, which deal with non-void methods. *TestAnalyzer* provides built-in return value generators for primitive types and objects. Further project-specific generators can be provided, which may support further special complex types.
- the method filters for excluding undesired methods (e.g., `ToStringMethodFilter`, or `SetterGetterFilter`)
- the number of threads that are concurrently analyzing different mutants. This allows reducing the analysis duration, but one needs to be sure that the test cases are thread-safe; that is, concurrently running test cases must not influence each other, as it may be the case when they write to the same temporary folder.
- a combination of constant and variable timeout values to restrict the analysis duration of a mutant.
- further extensions, for example, analyzers to collect metrics about methods and test cases.

#### 4.4.2 Alternative Implementation in Descartes

Based on the approach presented in [Niedermayr et al. \(2016\)](#), [Vera-Pérez et al. \(2018\)](#) implemented the proposed mutation operator in *Descartes*,<sup>1</sup>

<sup>1</sup> <https://github.com/STAMP-project/pitest-descartes>

which is a plug-in for the mutation testing tool PIT. The implementation as a plug-in allows it to benefit from the mature infrastructure provided by PIT. Thus, the plug-in can focus on the mutation operator and does not need to bother with the dependencies, build process, and used test framework of the system under test.

*Commonalities in both implementations:*

- Both implementations of the proposed mutation operator are implemented for Java and perform mutations on the byte code.
- Both tools support the test frameworks JUnit and TestNG.
- Both tools conduct a coverage analysis before the actual analysis to reduce the analysis effort by only considering methods that are covered by at least one test case and only executing for a mutated method those test cases that cover the method.
- Both tools examine mutants in a separate JVM process and apply time-outs to keep endless loops under control.
- The mutation operator of *Descartes* follows the concept presented in Section 4.3. However, methods that return an object are handled differently.

*Differences between both implementations:*

- Unlike PIT, the *TestAnalyzer* does not integrate with the build system. Therefore, it is necessary to provide the compiled artifacts of the application and test code along with the dependencies of the system under test.
- The *TestAnalyzer* does not stop the analysis of a mutant after it is killed by the first test case; thus, it enables a full mutation matrix to be computed. PIT was extended to enable that as well.
- The *TestAnalyzer* differentiates between methods that are covered by the actual test case and methods that are invoked during a test's set-up

or tear-down procedures. The mutation analysis is only conducted on methods that are invoked during the actual test execution. PIT does not draw this distinction.

- The *TestAnalyzer* and *Descartes* handle methods that return an object differently. The *Descartes* plug-in uses `null` as value for methods that return an object and an empty array for methods that return an array. The *TestAnalyzer* employs a factory to create an instance of a type; the factory combines a reflection approach with further dedicated logic to instantiate common types, and can be extended to support additional project-specific types.

The *TestAnalyzer* tool is used in the study in Chapter 5; *PIT* with *Descartes* is used in the empirical studies in Chapters 6 and 8.

## 4.5 Summary

Despite several optimizations, mutation testing using traditional mutation operators is computationally expensive and suffers from the problem of equivalent mutants. Unlike fine-grained mutation operators, the presented operator works at the method level and mutates methods by removing their whole logic. Thereby, this light-weight approach makes it possible to identify pseudo-tested methods; that is, methods that are covered solely by test cases that cannot detect faults in them. This mutation approach reduces the computational complexity by keeping the number of mutants low, addresses and avoids most equivalent mutants, and provides results that are easily comprehensible and actionable for developers.

We provide an implementation of the mutation operator in the tool *TestAnalyzer*. Vera-Pérez *et al.* (2018) developed a second, independent implementation, which allows researchers to mitigate the threat to validity regarding the mutation testing tool.

CHAPTER 

PRESENCE OF  
PSEUDO-TESTED METHODS  
AND THEIR CHARACTERISTICS

*This chapter shows that pseudo-tested methods are common and discusses their characteristics.*

## 5.1 Introduction

In this chapter, we want to shed light on pseudo-tested methods. To do so, we conduct a mutation analysis with the presented mutation operator on 19 study objects. We want to find out whether pseudo-tested methods exist in real projects and how common they are. We classify them by their functional purpose to estimate their relevance. Furthermore, we study whether pseudo-tested methods exhibit distinct characteristics, which make it possible to identify these methods using a static code analysis.

*The main contributions of this chapter are:* We conduct the first study on pseudo-tested methods and show that they are present in all study objects. We further show that pseudo-tested methods are relevant and should be tested more thoroughly. Finally, we identify measures that exhibit a moderate correlation with pseudo-testedness in some study objects.

An earlier version of the study presented in this chapter has been published in [Niedermayr et al. \(2016\)](#).

## 5.2 Study Objects

For this study, we selected and analyzed 19 open-source projects from GitHub<sup>1</sup> (GH), Google Code<sup>2</sup> (GC), and CQSE GmbH<sup>3</sup> (CQ). The projects satisfy the following criteria: They must use Java as programming language because the developed mutation testing tool is designed for Java and operates on Java byte code. Furthermore, they must contain a test suite, which uses either the *JUnit* or *TestNG* framework.

Table 5.1 lists the selected projects and their characteristics. The smallest project consists of 4,312 LOC, and the largest one has 443,092 LOC. The projects can be classified into libraries with unit tests and systems with

---

<sup>1</sup> <https://github.com>

<sup>2</sup> <https://code.google.com/archive>

<sup>3</sup> <https://www.cqse.eu/en>

Table 5.1: Study objects.

Project	LOC	# Tests	Line cov.	Origin
<i>Libraries with unit tests</i>				
APACHE COMMONS COLLECTIONS	109.4 k	4,372	81.6%	GH
APACHE COMMONS LANG	100.4 k	1,996	93.0%	GH
APACHE COMMONS MATH	275.6 k	3,427	84.8%	GH
APACHE COMMONS NET	53.6 k	163	29.0%	GH
ASTERISK	76.1 k	217	10.8%	GH
CONQAT ENGINE CORE	26.4 k	143	46.5%	CQ
CONQAT LIB COMMONS	39.7 k	611	59.4%	CQ
JABREF	124.1 k	1,561	34.4%	GH
JFREECHART	234.0 k	2,219	59.5%	GH
JSONDoc	4.3 k	26	81.2%	GH
TWITTER GRAPHJET	15.6 k	81	83.3%	GH
URBAN-AIRSHIP	27.8 k	575	86.8%	GH
<i>Systems with integration tests</i>				
CONQAT DOTNET	8.0 k	20	15.6%	CQ
DAISYDIFF	11.3 k	1 <sup>a</sup>	49.8%	GC
HISTONE	244.0 k	89	79.6%	GH
LITTLEPROXY	7.3 k	18	45.4%	GC
PREDICTOR	7.7 k	21	77.2%	GC
SYMJA	443.1 k	445	19.3%	GC
TSPMCCABE	45.0 k	10	40.6%	GC

<sup>a</sup> The test is parameterized and executed with 247 different input files.

integration tests. For systems, we identified integration tests based on the project's package structure and test class names.

### 5.3 Execution of the Mutation Analysis

To conduct the mutation analysis, we used the *TestAnalyzer*<sup>1</sup> tool.

We developed for each study object a tailored factory for creating instances of non-primitive types so that we can mutate most of the covered methods. The factories are based on the generic factory presented in Section 4.3 and support project-specific types that cannot be handled generically. For 7 study objects, the factories can create instances for more than 90% of the non-primitive return types used in methods. For the remaining 12 study objects, the factories support at least 80% of the used non-primitive return types.

In addition to methods that are generally not supposed to be mutated (see Section 4.3), we excluded in this study very simple setter and getter methods that consist of exactly one statement. We considered them as too trivial to be tested such that their mutation testing result is not of interest.

### 5.4 Existence of Pseudo-Tested Methods

Table 5.2 presents the number and proportion of pseudo-tested methods per project. Pseudo-tested methods are present in all projects. The proportion of pseudo-tested methods out of all mutated methods (*i.e.*, covered and not excluded methods) ranges between 1.1% and 42.3%. The mean proportion of pseudo-tested methods over all projects is 13.7% (median: 10.1%).

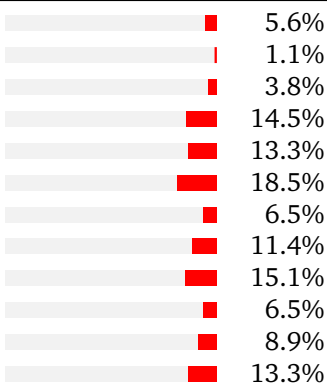
*Pseudo-tested methods are present in all study objects.*

It is striking that the proportion of pseudo-tested methods is clearly the highest in three systems with integration tests: in PREDICTOR with 42.3%, in LITTLEPROXY with 40.6%, and in SYMJA with 25.6%. If we differentiate between projects with unit tests and projects with integration tests, we observe that the mean proportion of pseudo-tested methods is 9.9% for unit tests and 20.2% for integration tests. The standard deviation is 5.3% and 15.9%,

<sup>1</sup> <https://github.com/cqse/test-analyzer>



Table 5.2: Pseudo-tested methods in the study objects.

Project	Pseudo-tested out of all <i>mutated</i> methods		
	#		%
APACHE COMMONS COLL.	125		5.6%
APACHE COMMONS LANG	19		1.1%
APACHE COMMONS MATH	126		3.8%
APACHE COMMONS NET	31		14.5%
ASTERISK	27		13.3%
CONQAT ENGINE CORE	58		18.5%
CONQAT LIB COMMONS	54		6.5%
JABREF	135		11.4%
JFREECHART	506		15.1%
JSONDOC	5		6.5%
TWITTER GRAPHJET	31		8.9%
URBAN-AIRSHIP	171		13.3%
CONQAT DOTNET	11		10.1%
DAISYDIFF	7		5.2%
HISTONE	30		8.5%
LITTLEPROXY	28		40.6%
PREDICTOR	91		42.3%
SYMJA	605		25.6%
TSPMCCABE	16		8.9%

respectively. However, there is no clear difference in the median (10.2% respectively 10.1%). This can also be seen in the boxplots in Figure 5.1. We conclude that the proportion of pseudo-tested methods can reach higher values in projects with integration tests and heavily varies among these projects. Consequently, code coverage from integration tests might not necessarily be a good approximation for test effectiveness and should be interpreted with caution.

Finally, Figures 5.2 and 5.3 visualize the number and extent of pseudo-tested methods in two study objects. In these treemaps (Shneiderman, 1992; Zörner, 2012), each rectangle represents a method and the size of a

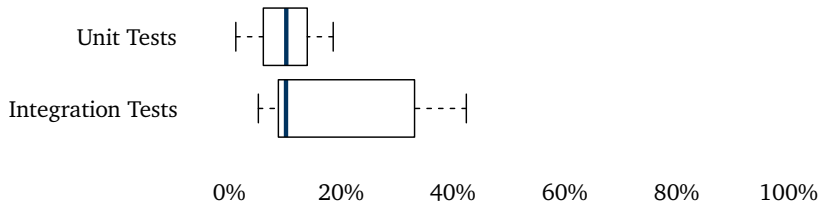


Figure 5.1: Boxplots comparing the proportion of pseudo-tested methods between projects with unit tests and projects with integration tests.

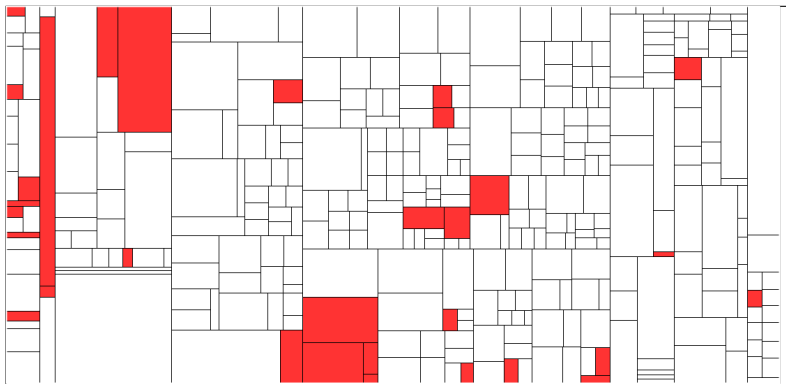


Figure 5.2: Distribution of pseudo-tested methods in TWITTER GRAPHJET.

rectangle corresponds to a method’s size in LOC. The rectangles are arranged according to the enclosing class and the package structure so that rectangles for methods in the same class are drawn next to each other. All rectangles colored in red represent pseudo-tested methods. The treemaps illustrate that pseudo-tested methods are spread across the whole code base; that is, these methods are not clustered in single parts of a project.

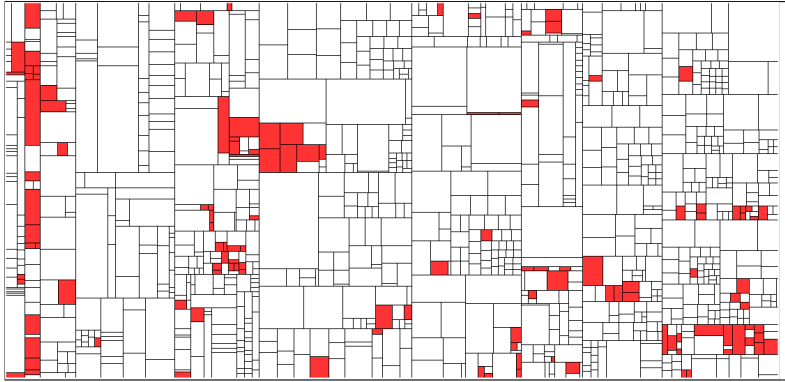


Figure 5.3: Distribution of pseudo-tested methods in JABREF.

## 5.5 Examples of Pseudo-Tested Methods

We discuss three examples of pseudo-tested methods that are present in the study objects. Many pseudo-tested methods are indirectly invoked through other methods and covered by more than a single test case. For better presentation, we only discuss methods that are directly and exclusively invoked by one test case.

Listing 5.1 presents a pseudo-tested method from the project JFREECHART. The method `setCategoryKeys(Comparable[] categoryKeys)`<sup>1</sup> takes an array of keys as input. It checks whether the array as such is not null and matches the required length, checks whether no entry in the array is null, stores the array, and notifies listeners about this operation.

This method is covered only by the similarly named test case in Listing 5.2 (`testSetCategoryKeys()`<sup>2</sup>), which directly invokes it.

---

<sup>1</sup> <https://github.com/jfree/jfreechart/blob/820773a5a2e84339f46bc79b77cc44092fb0d3be/src/main/java/org/jfree/data/category/DefaultIntervalCategoryDataset.java>

<sup>2</sup> <https://github.com/jfree/jfreechart/blob/820773a5a2e84339f46bc79b77cc44092fb0d3be/src/test/java/org/jfree/data/category/DefaultIntervalCategoryDatasetTest.java>

```

1 public void setCategoryKeys(Comparable[] categoryKeys) {
2     ParamChecks.nullNotPermitted(categoryKeys, "categoryKeys");
3     if (categoryKeys.length != getCategoryCount()) {
4         throw new IllegalArgumentException("The number of
5             ↪ categories does not match the data.");
6     }
7     for (int i = 0; i < categoryKeys.length; i++) {
8         if (categoryKeys[i] == null) {
9             throw new IllegalArgumentException(
10                 ↪ "DefaultIntervalCategoryDataset." +
11                 ↪ "setCategoryKeys(): " + "null category not
12                 ↪ permitted.");
13         }
14     }
15     this.categoryKeys = categoryKeys;
16     fireDatasetChanged();
17 }

```

Listing 5.1: Example of a pseudo-tested method in JFREECHART.

```

1 @Test
2 public void testSetCategoryKeys() {
3     DefaultIntervalCategoryDataset empty = new
4         ↪ DefaultIntervalCategoryDataset(new double[0][0],
5         ↪ new double[0][0]);
6     boolean pass = true;
7     try {
8         empty.setCategoryKeys(new String[0]);
9     }
10    catch (RuntimeException e) {
11        pass = false;
12    }
13    assertTrue(pass);
14 }

```

Listing 5.2: Test case covering the pseudo-tested method from Listing 5.1.

The test case creates a new instance of the class under test and invokes the `setCategoryKeys(...)` method on it. It will pass as long as this method

does not throw an exception of type `RuntimeException`. This test case is presumably designed to ensure that the method allows an invocation with an empty array. To be precise, it will also ensure that the invocation of `fireDatasetChanged()` does not result in an exception. Besides that, the test case does not check anything else and cannot detect if the whole logic is missing. Because of that and since no other test case covers the method, this method is pseudo-tested and its code coverage result is misleading. Since the test does not detect our invasive mutation, it will also not be able to detect subtler faults in the method (e.g., incomplete array validations, a missing or inappropriate assignment of the array to the instance variable, or a missing listener notification).

Another pseudo-tested method is presented in Listing 5.3. The method `createMD5Digest(Collection<String> bases)`<sup>1</sup> from the project `CONQAT LIB COMMONS` takes a collection as input, sorts the collection items, joins them to a single value, and returns the thereof computed MD5 digest.

```
1 public static String createMD5Digest(Collection<String> bases) {  
2     List<String> sortedBases = CollectionUtils.sort(bases);  
3     return createMD5Digest(StringUtils.concat(sortedBases,  
4         ↪ StringUtils.EMPTY_STRING));  
5 }
```

Listing 5.3: Example of a pseudo-tested method in `CONQAT LIB COMMONS`.

The test case in Listing 5.4<sup>2</sup> invokes this method twice with a collection holding the same elements in a different order. The test then compares whether both invocations result in the same MD5 value; that is, it intends to ensure that the computation of the MD5 digest is not sensitive to the order of the collection's elements. The way this test is implemented, it uses the results of the method invocations as source for both the expected value (test oracle) and the observed value (method output).

<sup>1</sup> <https://www.cqse.eu/download/conqat/conqat-source-2015.2.zip/org/conqat/lib/commons/digest/Digester.java>

<sup>2</sup> <https://www.cqse.eu/download/conqat/conqat-source-2015.2.zip/org/conqat/lib/commons/digest/DigesterTest.java>

Despite code coverage indicating that this method is fully tested, the test will not detect any other faults besides the mentioned one. For a method like this one that returns a string value, the applied mutation operator removes the whole logic from the method and creates two mutants. The first mutant will return an empty string. When testing this mutant, the test case will compare the two empty strings and pass because of their equality. The second mutant will return a string consisting of the letter “A” and the test case comparing “A” with “A” will also pass. Consequently, the mutation will not be detected such that the method is to be considered as pseudo-tested.

```
1 @Test
2 public void testCreateFingerprintFromCollection() {
3     String digest1 = Digester.createMD5Digest(
4         ↪ Arrays.asList(new String[] { "a", "b", "c" }));
5     String digest2 = Digester.createMD5Digest(
6         ↪ Arrays.asList(new String[] { "c", "b", "a" }));
7     assertEquals(digest1, digest2);
8 }
```

Listing 5.4: Test case covering the pseudo-tested method from Listing 5.3.

A third example of a pseudo-tested methods is presented in Listing 5.5. The method `getSortedValues()`<sup>1</sup> from the APACHE COMMONS MATH project is designed to retrieve previously set values and to sort them.

```
1 public double[] getSortedValues() {
2     double[] sort = getValues();
3     Arrays.sort(sort);
4     return sort;
5 }
```

Listing 5.5: Example of a pseudo-tested method in APACHE COMMONS MATH.

---

<sup>1</sup> <https://github.com/apache/commons-math/blob/724795b5513651e1e34fae3904d1b58229ce9c17/src/main/java/org/apache/commons/math3/stat/descriptive/DescriptiveStatistics.java>

This method is covered only by the test case `testGetValues()`<sup>1</sup> in Listing 5.6, which directly invokes it. Besides other statements and assertions in this test case targeting at other methods, this test case retrieves an array by invoking the `getSortedValues()` method. It then iterates over the array and checks each entry using the `assertEquals` method.

```
1 @Test
2 public void testGetValues() {
3     DescriptiveStatistics stats = createDescriptiveStatistics();
4     for (int i = 100; i > 0; --i) {
5         stats.addValue(i);
6     }
7     int refSum = (100 * 101) / 2;
8     Assert.assertEquals(refSum / 100.0, stats.getMean(), 1E-10);
9     double[] v = stats.getValues();
10    for (int i = 0; i < v.length; ++i) {
11        Assert.assertEquals(100.0 - i, v[i], 1.0e-10);
12    }
13    double[] s = stats.getSortedValues();
14    for (int i = 0; i < s.length; ++i) {
15        Assert.assertEquals(i + 1.0, s[i], 1.0e-10);
16    }
17    Assert.assertEquals(12.0, stats.getElement(88), 1.0e-10);
18 }
```

Listing 5.6: Test case covering the pseudo-tested method from Listing 5.5.

The mutated method will return an empty array. Although the test case checks the array entries, the assertion statement will not be reachable for an array of length 0 because the body of the loop will never be executed. Consequently, this test case will not kill the mutant such that the method is pseudo-tested.

---

<sup>1</sup> <https://github.com/apache/commons-math/blob/724795b5513651e1e34fae3904d1b58229ce9c17/src/test/java/org/apache/commons/math3/stat/descriptive/DescriptiveStatisticsTest.java>

## 5.6 Characteristics of Pseudo-Tested Methods

In this section, we study pseudo-tested methods to learn more about their characteristics.

**RQ 1: What functional category do the pseudo-tested methods belong to?** This research question sheds light on the functional purpose of pseudo-tested methods. We want to know what kinds of methods are pseudo-tested to estimate how severe their lack of test effectiveness is for a project. Some methods may not warrant being tested because they only concern logging or performance optimizations, but others may be relevant and need more thorough testing to avoid (regression) faults.

To approach this research question, we first inspected a couple of pseudo-tested methods and defined functional categories, which characterize the purpose of a method. Then, we manually assigned exactly one category to each of the 2,076 pseudo-tested methods. We conducted the classification based on a method's name; when we were unsure about a method, we additionally consulted the method's code. Finally, we asked another developer, which was not involved in this study, to classify 100 randomly sampled pseudo-tested methods based on the category descriptions, and computed the inter-rater agreement.

The functional categories are as follows. Table 5.3 presents an example of each category.

- *non-deterministic computations*: This category comprises methods that are considered to behave differently at different invocations with the same input and state. An example is a method that generates a random number. It is difficult to properly test such methods; therefore, we assume that they are likely pseudo-tested when covered.
- *monitoring*: This category comprises methods for logging information about the program execution or producing other debugging output.
- *optimization*: Methods in this category intend to reduce computational



Table 5.3: Functional method categories.

Functional category	Examples
non-deterministic	<code>nextRandomInt()</code>
monitoring	<code>logInfo(String)</code> , <code>dumpMemoryUsage()</code>
optimization	<code>addToCache(Object)</code>
validation	<code>checkIndex(int)</code> , <code>validateParam(int)</code>
finalization	<code>finalize()</code> , <code>closeStream()</code>
toString	<code>toString()</code>
event handling	<code>notifyListeners()</code> , <code>fireValueChange()</code>
object creation	<code>createAlgorithm(String)</code> , <code>newNode(Object)</code>
transformation	<code>formatDate(Date)</code> , <code>emptyIfNull(String)</code>
preparation	<code>initWorkflow()</code> , <code>setUpBlock()</code>
setter & getter	<code>isValid()</code> , <code>getX(int)</code>
object identity	<code>equals(Object)</code> , <code>compareTo(T)</code>
application logic	<code>computeLSB()</code> , <code>solvePhase1()</code>

efforts by avoiding redundant computations (for example, by caching results).

- *validation*: Methods that check the validity of variable values belong to this category. For example, a method in this category may check whether a numeric variable's value is within a certain range.
- *finalization*: This category contains methods that perform tear-down operations after a completed action. For example, they may flush a stream or terminate a connection.
- *toString*: Java's `toString` methods, which provide a string representation of an object, form this category. Their result is often only used for debugging purposes, but it can also be used to represent data (e.g., in the class `StringBuilder`), which is further processed or displayed to the user.
- *event handling*: This category comprises methods that notify listeners about events when they occur.
- *object creation*: Methods that create and initialize an instance of a type belong to this category. Such methods usually determine the class that should be used for instantiation based on an input value and the execution context.
- *transformation*: Methods that format, adjust, or convert a value are in this category.
- *preparation*: This category contains methods that perform set-up operations; that is, methods conducting initializations to prepare an environment, a computation, or another action.
- *setter & getter*: This category contains non-trivial setters and getters, which assign and retrieve a value, respectively. Note that very simple setters and getters consisting of a single statement were omitted from the mutation analysis (see Section 5.3). Therefore, these methods contain further logic. For example, a non-trivial setter may validate a value before assigning it to a field.

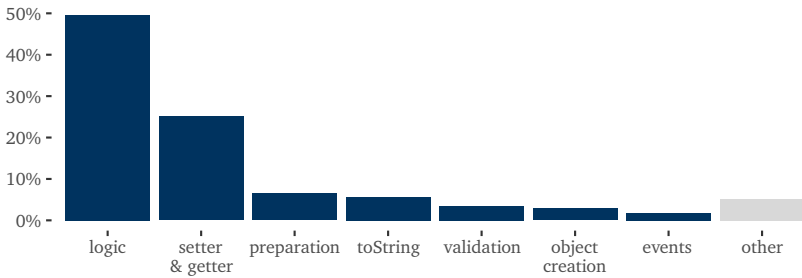


Figure 5.4: Classification of pseudo-tested methods by their functional category.

- *object identity*: This category includes equals methods, which check whether an object is semantically equal to another one, and compareTo methods, which determine the natural order of two objects. Note that hashCode methods, which also concern an object’s identity, were not in the analysis scope.
- *application logic*: This category contains methods that concern core aspects of an application, such as, computation logic or data handling.

The listed categories were ordered by our perceived severity so that methods in the first categories are assumed to be generally less likely to affect the correctness of the program execution than the latter ones. For example, methods that close a stream or dump the current memory consumption should ideally not be pseudo-tested; however, if they are pseudo-tested and contain undiscovered faults, their consequences are likely less severe than of other methods with a higher influence on the program execution.

Figure 5.4 presents the distribution of pseudo-tested methods per functional category aggregated over all study objects. We can see that the majority of pseudo-tested methods (49.5%) belongs to the category “application logic.” Further, 25.0% belong to non-trivial “setters and getters.” Thereby, these two categories already account for 74.5% of all pseudo-tested methods.

We conclude that pseudo-tested methods concern relevant methods such that their lack of test effectiveness is a relevant problem. The inter-rater agreement measured with Cohen's kappa was 0.76. According to Landis and Koch (1977), this indicates a substantial agreement between the raters.

Vera-Pérez *et al.* (2017) also confirmed in their study that developers consider flaws in the test suite uncovered by pseudo-tested methods as relevant. However, according to them, developers do not always give improving tests the highest priority because this activity requires time and effort, and is in a resource conflict with bugfixing and development tasks.

*In these study objects, the majority of pseudo-tested methods can be considered as relevant and should be tested more thoroughly.*

**RQ 2: How do properties of pseudo-tested methods differ from other methods?** With this research question, we want to find differences between pseudo-tested methods and the remaining covered methods. If we identify distinctive properties of pseudo-tested methods, we could exploit them to reveal pseudo-tested methods with a static code analysis.

To answer this research question, we computed measures about covered methods and their relationships to test cases. We then applied statistical tests to analyze whether a method's mutation testing verdict (pseudo-tested or not) is correlated with these measures. In particular, we studied whether the mutation testing verdict is correlated with:

- a method's *return type*. We differentiated between “void,” “primitive and string,” and “object.”
- a method's *length* in number of statements.
- a method's *line or branch coverage*. We used JaCoCo<sup>1</sup> to compute the coverage. For covered methods without branches, we used 100% as branch coverage value.

---

<sup>1</sup> <https://www.eclemma.org/jacoco>

- a method's *number of covering test cases*.
- the *minimal scope of a method's covering test cases*. The scope of a test case corresponds to the number of methods that it covers. We want to know whether methods that are covered solely by test cases that cover many methods are more likely pseudo-tested than methods that are covered by more “target-oriented” test cases.

The mutation testing verdict is on the nominal scale. The method return type is also on the nominal scale, and all other measures are numeric. We applied the Chi-squared test to analyze the correlation with the return type (hypothesis  $H_0$ : mutation testing verdict and return type are independent). For all other measures, we applied Spearman's rank correlation coefficient because the numeric values are not normally distributed (hypothesis  $H_0$ : mutation testing verdict and measure are independent). We used a significance level of 5%.

For the statistical tests, we excluded three projects containing fewer than 15 pseudo-tested methods (DAISYDIFF, JSONDOC, and CONQAT DOTNET). We excluded three further projects from the Chi-squared test due to the test's requirement that the expected count is at least five for each cell in the contingency table (APACHE COMMONS LANG, HISTONE, and TSPMCCABE).

The results of the statistical tests are:

- *return type*: The return type of a method has a statistically significant influence on the mutation testing verdict in 9 out of 13 projects. Void methods are more likely pseudo-tested. We believe that test cases can more easily verify a return value than a changed state such that faults influencing a return value might be simpler to reveal. Figure 5.5 presents the proportion of pseudo-tested methods out of the mutated methods by return type.
- *method length*: This correlation is significant in 9 of 16 projects. The rho-value is +0.38 for SYMJA and +0.27 for CONQAT ENGINE CORE, indicating that longer methods are more likely to be effectively tested

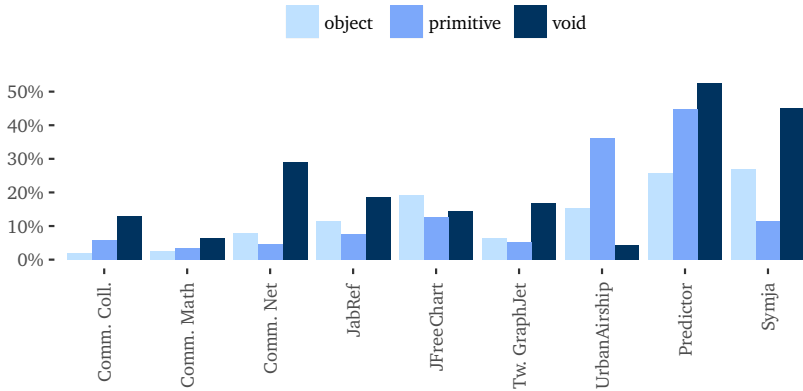


Figure 5.5: Proportion of methods that are pseudo-tested out of the mutated methods per return type.

in respect to this mutation operator (*i.e.*, less likely pseudo-tested). However, the rho-value is below +0.20 for all other projects. It is surprising that this correlation is not stronger because we consider our mutation operator to be more severe for longer methods since it removes “more logic” in those methods.

- *line or branch coverage*: No statistically significant correlation can be determined. The correlation of the mutation testing verdict with a method’s line coverage is in only 5 of 16 projects significant (branch coverage: 6 of 16). Furthermore, the rho-values do not point towards a distinct direction. An investigation of the coverage values showed that 79.7% of the pseudo-tested methods exhibit full line coverage, and 74.4% have full branch coverage.
- *number of covering test cases*: The results suggest that methods are less likely pseudo-tested if they are covered by many test cases. This correlation is statistically significant in 14 of 16 projects. The rho-value of the correlation exceeds +0.20 in 5 projects; it reaches +0.42

Table 5.4: Results of the correlation tests. The p-value expresses the significance of the correlation. The rho-value expresses the strength of linkage and ranges between  $-1$  and  $+1$ . p-values  $< 0.05$  and absolute rho-values  $\geq 0.2$  are highlighted.

Project	Length		Line coverage		Branch coverage		# Covering tests		Min. test scope	
	p-value	rho	p-value	rho	p-value	rho	p-value	rho	p-value	rho
APACHE COMMONS COLL.	0.146	-0.03	<0.001	+0.09	<0.001	+0.10	0.014	-0.05	<0.001	-0.12
APACHE COMMONS LANG	0.022	+0.06	0.596	-0.01	0.253	+0.03	0.507	+0.02	0.001	-0.08
APACHE COMMONS MATH	<0.001	+0.10	0.047	+0.03	0.365	+0.02	<0.001	+0.06	<0.001	-0.11
APACHE COMMONS NET	0.644	+0.03	0.341	+0.07	0.028	+0.15	<0.001	+0.22	0.452	-0.05
ASTERISK	0.010	+0.18	0.077	-0.12	<0.001	-0.24	0.013	-0.17	0.010	-0.18
CONQAT ENGINE CORE	<0.001	+0.27	0.697	-0.02	0.855	+0.01	<0.001	+0.26	<0.001	-0.19
CONQAT LIB COMMONS	<0.001	+0.16	0.135	+0.05	0.527	+0.02	0.006	+0.09	0.154	-0.05
JABREF	<0.001	+0.11	0.558	+0.02	0.830	+0.01	<0.001	+0.20	<0.001	-0.31
JFREECHART	<0.001	-0.16	<0.001	+0.22	<0.001	+0.25	<0.001	-0.07	<0.001	-0.48
TWITTER GRAPHJET	0.947	$\pm 0.00$	0.487	+0.04	0.810	-0.01	<0.001	+0.26	<0.001	-0.29
URBANAIRSHIP	<0.001	+0.09	0.021	-0.06	0.024	-0.06	0.030	+0.06	<0.001	-0.18
<i>unit test projects</i>	0.362	-0.01	<0.001	+0.11	<0.001	+0.12	0.038	+0.02	<0.001	-0.26
HISTONE	0.158	+0.08	0.776	-0.02	0.989	$\pm 0.00$	0.009	-0.14	0.002	+0.17
LITTLEPROXY	0.143	+0.18	0.256	-0.14	0.917	+0.01	0.262	+0.14	<0.001	-0.43
PREDICTOR	0.145	+0.10	0.058	-0.13	0.255	-0.08	<0.001	+0.28	<0.001	-0.48
SYMJA	<0.001	+0.38	<0.001	-0.26	<0.001	-0.29	<0.001	+0.42	<0.001	-0.53
TSPMCCABE	0.077	+0.13	0.595	-0.04	0.074	+0.13	0.039	+0.15	0.274	-0.08
<i>integration test projects</i>	<0.001	+0.33	<0.001	-0.23	<0.001	-0.23	<0.001	+0.34	<0.001	-0.39

in SYMJA. Nevertheless, a lot of methods exist that are not pseudo-tested, although they are covered only by a few test cases.

- *minimal scope of covering test cases*: Methods covered solely by test cases that invoke many methods are more likely pseudo-tested; that is, the risk of pseudo-testedness for a method decreases with a lower minimal scope of its covering test cases. This negative correlation holds in 13 out of 16 projects. Rho-values reach  $-0.53$  and go below  $-0.20$  in 4 further projects. Therefore, this measure might be useful in some projects to predict pseudo-tested methods.

Table 5.4 contains the results of the Spearman's correlation tests separately for each projects. It also contains the results aggregated over all projects with unit tests and all projects with integration tests.

*The results indicate in some projects that methods are less likely pseudo-tested if they are covered by many test cases or by test cases that cover only few other methods.*

## 5.7 Threats to Validity

We separated the threats into internal and external validity. Threats to internal validity comprise reasons why the results could be invalid for the study objects. Threats to external validity concern the generalization of the study results.

*Threats to internal validity*: To conduct the mutation analysis, we used the *TestAnalyzer* tool. Its source code is publicly available on GitHub.<sup>1</sup> We developed this tool with great care, implemented automated unit and integration tests, and manually verified the results of many code samples. Nevertheless, the implementation could still contain faults that affect analysis results. [Vera-Pérez et al. \(2018\)](#) developed *Descartes*, which is a plug-in

<sup>1</sup> <https://github.com/cqse/test-analyzer>



for PIT and also implements the mutation operator to determine pseudo-testedness. Using their implementation, [Vera-Pérez et al.](#) replicated our first study presented in [Niedermayr et al. \(2016\)](#). They confirmed our results by showing that pseudo-tested methods also exist in all their study objects.

Another threat to internal validity is that some methods considered pseudo-tested might actually be equivalent mutants. The design of the mutation operator reduces the likelihood of equivalent mutants due to the invasive transformation and the use of two mutants for methods returning a primitive value (see [Section 4.2](#)). Furthermore, empty methods were excluded from the analysis. Therefore, we consider this threat to be negligible.

[Vera-Pérez et al. \(2017\)](#) pointed out that the return values used by the mutation operator might influence whether a mutant is killed or not. However, in most cases this should not influence the results when two different mutants are created.

Non-deterministic tests, especially flickering ones, are a threat to validity. When such test cases had failed during the initial analysis, they were excluded, which caused the test suite to shrink. When they had passed during the initial analysis and (sometimes) failed during the mutant analysis, they might have killed mutants of methods that are actually pseudo-tested. Flickering tests are often a problem in tests at higher levels (*e.g.*, in UI tests), but less common in unit tests. A mitigation strategy could be to execute tests more than once in the initial analysis.

A threat with similar consequences as with non-deterministic tests is caused by tests that initially pass but begin to repeatedly fail when the environment is not properly reset after a test execution. This may occur, for example, when the execution of mutated code results in modifications of existing files or when failing tests leave temporary files behind. Even if this problem applies only to a few tests, it can influence mutation results because a single failing test is enough for a method to be not pseudo-tested. As with the previous threat, this threat concerns the mutation analysis and is not specific to our mutation operator. Future work is necessary to determine the relevance of this threat.

Study objects that contain test cases that fail on the original code are a

further threat to internal validity. Failing test cases can occur because of the test environment or faulty code in the study objects. Some test cases may rely on further data stored in files, databases with certain data models and contents, other connected systems, or the network connection. Despite attempts to supply all needed files in the test execution folder and set up required databases according to the project documentation, some test cases still failed. This was the case in `APACHE COMMONS NET`, which presumably required certain firewall settings for some of its test cases. Failing test cases were excluded from the analysis. If the excluded test cases had worked, they could 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 when nearly all test cases passed on the original code.

The selection of integration tests in systems is another threat to internal validity. The corresponding test classes were identified based on accordingly named source folders, packages, and classes. However, different projects may have different perceptions of integration tests.

Regarding RQ 1, we assigned a functional category to a pseudo-tested method mostly by considering a method's name. Even though a method's code was inspected in case of uncertainties, it was not feasible to inspect all pseudo-tested methods due to their number. Therefore, some methods might actually belong to a functional category other than the assigned one. To avoid bias, we compared our classification on a random sample of 100 pseudo-tested methods with a second classification conducted by a developer not involved in the study. The inter-rater agreement yielded a Cohen's kappa value of 0.76.

*Threats to external validity:* A threat to external validity is that the results of the selected open-source projects might not be applicable to other projects (e.g., closed-source systems). We tried to mitigate this threat by considering several projects with different characteristics and application domains as study objects. This threat is further mitigated by [Vera-Pérez et al. \(2017\)](#), who replicated our study with 17 additional open-source projects. Further

studies are still necessary to determine whether the results apply to closed-source projects. According to [Petrović and Ivanković \(2018\)](#), mutation testing results are affected by the programming language; therefore, the same applies to projects in other languages than Java.

## 5.8 Summary

We conducted a mutation analysis to identify pseudo-tested methods and analyzed the resulting methods. Pseudo-tested methods are present in all 19 open-source projects. The proportion of pseudo-tested methods out of the mutated covered methods varies among projects and is between 1.1% and 42.3%. Higher proportions are achieved in projects with integration tests. Pseudo-tested methods are reported to be covered by test cases, but this is misleading and gives a false sense of effectiveness, since tests cannot find faults in them.

Nearly 75% of the pseudo-tested methods concern the application logic or are non-trivial setters and getters; therefore, we consider their lack of test effectiveness to be a relevant problem.

Finally, we identified correlations in some projects, which indicates that methods are less likely to be pseudo-tested if they are covered by many test cases or by test cases that cover only few other methods. The next chapter looks at whether a machine-learning model, which combines several measures, can predict pseudo-tested methods without executing a mutation analysis.



CHAPTER  
6

# PREDICTING PSEUDO-TESTED METHODS

*This chapter introduces the minimal stack distance as a measure to describe the proximity between test cases and methods. It presents a machine-learning classifier that uses this measure, along with further easily computable measures, and shows that pseudo-tested methods can successfully be predicted using this classifier.*

## 6.1 Motivation

Even when it focuses on pseudo-tested methods, mutation testing may be too expensive or not applicable due to technical reasons in large, complex software systems (Jia and Harman, 2011). At the same time, code coverage is not necessarily meaningful enough to assess test suites (Inozemtseva and Holmes, 2014). Therefore, this chapter examines whether pseudo-tested methods can be predicted using a machine-learning model based on metrics, which characterize methods, test cases, and their relationships. The aim is to reduce the effort involved in identifying pseudo-tested code.

The underlying assumption for predicting pseudo-tested methods is that a test case that directly invokes a method is more likely to detect faults in it than another test case that accesses this method indirectly through many other methods. In other words, we assume that a method that is solely invoked by distant test cases is more likely to be pseudo-tested. The rationale behind this assumption is that a faulty state needs to be propagated through many methods until it reaches the test case, which contains the assertions. For methods invoked solely by distant test cases, the attainment of the RIPR model's reachability criterion (see Section 2.2) is impeded.

To operationalize this assumption, we propose a new measure called *minimal stack distance*, which expresses how close a test case comes to a given method. The present chapter defines this measure and describes its computation. It also studies whether a correlation exists between a method's minimal stack distance to all test cases and its mutation testing verdict, which expresses whether the method is pseudo-tested or not. Finally, we train a machine-learning model using stack distance values along with further measures, which can be collected in a single execution of a test suite, and evaluate the model's performance. The results suggest that such prediction models can be an alternative to mutation testing in scenarios where mutation testing takes too long or is not applicable due to other (technical) reasons. For example, prediction models could be employed in a continuous-integration pipeline.

The main contributions of this chapter are: First, we propose and study the minimal stack distance measure, which characterizes the proximity of a method to any of its test cases. Second, we evaluate a machine-learning classifier based on test-case method characteristics and show that classifiers to predict a method’s mutation testing verdict can come into question as an alternative to mutation testing or as a preceding, less costly step.

Parts of the content of this chapter have been published in [Niedermayr and Wagner \(2019\)](#).

## 6.2 Definitions

We define the *minimal stack distance between a method  $m$  and a test case  $t$*  as the length of the shortest path from  $t$  to  $m$  on the dynamic call graph.<sup>1</sup> The value is 1 for a method that is directly invoked by a given test case and, for example, 2 for a method that is indirectly invoked by a given test case through one other method.

We define the *minimal stack distance of a method  $m$*  as the shortest distance on the dynamic call graph between  $m$  and any of its covering test cases  $t \in T : cov_t(m)$ . It corresponds to the minimal distance on the call stack between the method  $m$  and all test cases. Figure 6.1 illustrates an example.

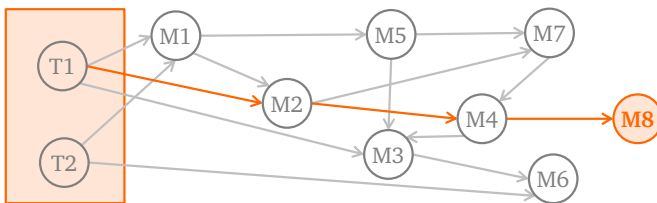


Figure 6.1: The minimal stack distance of method  $M8$  is 3. No test case can access  $M8$  through fewer method invocations.

<sup>1</sup> We define and apply minimal stack distance based on *methods*. However, the definitions are also applicable to *functions* in non-object-oriented programming languages.

A method is considered *covered* if it is executed by at least one test case. The *mutation testing verdict* of a covered method can either take the value *pseudo-tested* or *tested*. We also use common mutation testing terms explained in Section 2.4.1.

### 6.3 Approach

The following describes the computation of the minimal stack distance for Java applications, although this measure is also applicable to other programming languages. The steps to compute the minimal stack distance comprise the instrumentation of the code, the replacement of Java’s Thread class, and the recording of method invocations during the test execution. Figure 6.2 presents an overview of the computation.

- (1) *Instrumentation*: Each method of the application code needs to be instrumented so that the stack-recorder class is notified when a method is entered and exited. To instrument a method, we inserted a statement at the beginning of the method, which calls the recorder class with the signature of the considered method. A further statement is needed, which informs the recorder that the method invocation needs to be removed from the current stack when the method is left. To ensure that this statement is executed for each execution path, we introduced

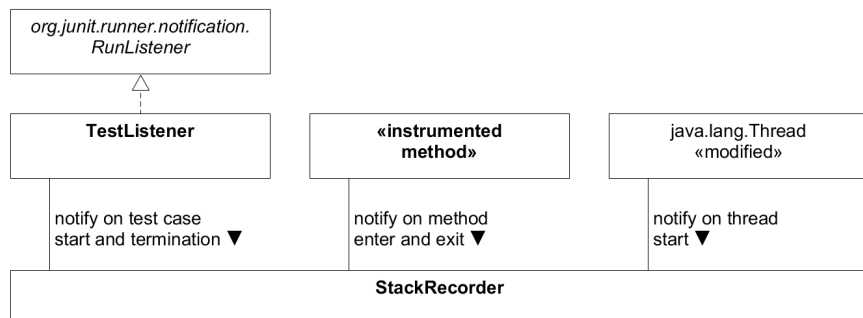


Figure 6.2: Overview of the stack distance computation.



a try-finally block, moved the original code into the try block, and inserted the statement into the new finally block. The finally block is always invoked when the method is left (even if an exception is raised or propagated).

To conduct the code instrumentation, we developed a Maven plug-in, which operates at the byte-code level and uses the ASM<sup>1</sup> library. The decompiled source code of an instrumented method might look as follows:

```
1 public int getSize() {
2     StackRecorder.push("org.SampleClass.getSize()");
3     try {
4         /* BEGIN ORIGINAL CODE */
5         return this.size;
6         /* END ORIGINAL CODE */
7     } finally {
8         StackRecorder.pop("org.SampleClass.getSize()");
9     }
10 }
```

Listing 6.1: Example of an instrumented method.

- (2) *Thread class replacement*: Tests may cover code that is concurrently executed. To avoid that the stack recorder puts method invocations from concurrent threads onto the same stack, the computation of the minimal stack distance needs to be thread-aware. This means that the current stack height needs to be recorded separately for each thread. To achieve that, it is necessary to be notified when a new thread is started and know the dependencies between the threads. Since Java's Thread class does not provide the possibility to register listeners, we took the original code from the Java Development Kit (JDK) and adjusted it so that the stack-recorder class is informed about the start of a new thread. We compiled the modified thread class and put it into the JDK's "endorsed" folder, which allows replacing existing

---

<sup>1</sup> <http://asm.ow2.io>

Java classes. The replacement of the thread class does not influence test results.

- (3) *Recording*: The final step is to execute the test suite and record the distances between test cases and methods. We used Maven's Surefire plug-in for the execution of unit tests and Failsafe plug-in for integration tests, and registered the stack-recorder class as a test listener in these plug-ins. Hence, the recorder will be notified when a new test case execution begins and can assign all subsequent method invocations to that test case. When a test case execution starts and an instrumented method is entered, the method's signature is pushed onto the recorder's stack of the current thread. The stack's height is then counted and, if appropriate, the distance from the executed test case to the start of the current thread is added. If the resulting distance constitutes a new minimum for a given test-case method pair, the pair's minimal stack distance value is updated. When an instrumented method is left, its signature is removed from the stack of the appropriate thread.

If a method is invoked recursively, the height of the stack increases with each invocation; however, we are only interested in the *minimal* stack distance of each test-case method pair.

In short, the recorder class holds the so-far minimal stack distance of each executed test-case method pair, the method invocations on the stack of each thread, and the relations between the threads. At the end of each test case execution, the minimal stack distance values are retained.

Another imaginable approach to compute the stack height by requesting the current thread to dump its stack trace (as done when creating exceptions) is not fast enough to viably do the computation in test executions.

The presented approach has the following limitations. We applied the instrumentation to all methods except constructors, which we excluded because they are difficult to instrument in such a way that their beginning is correctly intercepted. This is due to the fact that a constructor's very first

statement unavoidably delegates to another constructor or a super constructor such that the code there gets executed first. Consequently, constructor invocations will not be counted when computing the stack distance. Nevertheless, methods invoked by constructors are still considered. Furthermore, external libraries are not instrumented, which means that the method invocations in external libraries are not counted. The consequence of both limitations is that the computed stack distance will, in some cases, be slightly lower than the actual distance. Hence, the computed minimal stack distances should be considered as a lower bound.

## 6.4 Study

This section reports on the design and results of the empirical study that investigated the minimal stack distance and examined how well the mutation testing verdict of a method can be predicted.

### 6.4.1 Research Questions

The following research questions were investigated:

**RQ 1: Are methods with a higher minimal stack distance to the test cases more likely to be pseudo-tested?** The purpose of this research question is to find out whether the minimal stack distance of a method is correlated with the mutation testing verdict expressing whether the method is pseudo-tested or not. We hypothesized that a test case that never comes close to a given method is not effective in detecting faults in that method. Consequently, we expected that a method tested only by distant test cases would be less effectively tested; in other words, that methods with a high minimal stack distance are more likely to be pseudo-tested. The answer to this question helps determine whether stack distance can be a useful predictor of pseudo-tested methods.

**RQ 2: How well can the mutation testing verdict be predicted using**

**test-relationship measures?** Since mutation testing is costly, we wanted to find out whether a light-weight approach could approximate results gained from a mutation analysis. We were interested in predicting a method's mutation testing verdict based on measures characterizing relationships between methods and test cases. If such a prediction approach works well, it could be used as an alternative to mutation testing or as a preceding, less costly step.

#### 6.4.2 Study Objects

We selected study objects from GitHub<sup>1</sup> based on the following criteria. The projects need to be written in Java, contain test cases designed for the JUnit test framework, and use Maven as build system. We manually selected five Apache projects (COMMONS GEOMETRY, COMMONS IMAGING, COMMONS LANG, COMMONS MATH, COMMONS STATISTICS), and JFREECHART, which are popular open-source projects used in several empirical test studies (e.g., in Hemmati (2015), Inozemtseva and Holmes (2014), and Just *et al.* (2014b)). We selected additional study objects that satisfy the previously mentioned criteria by searching GitHub for recently updated projects with at least five forks (to require a certain popularity). We excluded a project if it was not possible to build it (e.g., due to compilation problems or unresolvable dependencies), if more than 5% of the test cases failed in a local execution of the original test suite, or if the mutation analysis was not successful (e.g., due to special test runners or class-loading mechanisms).

The selected study objects are from different domains and contain single- and multi-module Maven projects. Their characteristics are presented in Table 6.1. *LOC* (lines of code) refers to the application code (*i.e.*, code without test and sample code) and was measured with *Teamscale* (Haas *et al.*, 2019c). *# Tests* refers to the number of test cases as reported by Maven. *Line* and *branch coverage* were computed with JaCoCo.<sup>2</sup> The largest project, BIOJAVA, consists of 240.6 k LOC. COMMONS MATH contains with

---

<sup>1</sup> <https://github.com>

<sup>2</sup> <https://www.eclemma.org/jacoco>

Table 6.1: Study objects.

Name	Purpose	LOC	# Tests	Line cov.	Branch cov.	Git revision
APACHE COMMONS GEOMETRY	geometric utilities	19.4k	643	76.9%	70.7%	be34ad93
APACHE COMMONS IMAGING	image library	48.4k	575	71.3%	58.9%	eb98398b
APACHE COMMONS LANG	utility classes for Java	77.0k	4,053	95.0%	91.1%	1f0dfc31
APACHE COMMONS MATH	mathematics library	186.3k	5,254	89.8%	84.8%	ea fb16c7
APACHE COMMONS STATISTICS	statistics library	6.1k	358	91.5%	87.6%	aa5cbad1
BIOJAVA	biological data processing	240.6k	1,181	40.5%	38.5%	523c78e1
BITCOINJ	Java Bitcoin library	59.1k	5,222	67.5%	61.3%	911f6d49
GEOMETRY-API-JAVA	spatial data processing	87.0k	408	71.6%	59.4%	3704c220
GOOGLE-GSON	JSON serialization	14.8k	1,039	84.4%	79.2%	57085d62
GOOGLE HTTP JAVA CLIENT	HTTP client library	30.1k	635	54.9%	58.8%	df0e9f2a
GRAPHHOPPER	route planning library and server	60.5k	1,680	65.4%	60.9%	e954f008
JACKSON-DATABIND	databinding for JSON data	103.0k	2,159	77.8%	70.7%	bf604125
JAVAPARSER	parser and AST for Java	118.4k	1,284	59.8%	48.1%	1cca4c46
JFREECHART	chart library	222.8k	2,175	55.5%	46.4%	39df ee3c
JSOUP	HTML and CSS parser	18.2k	671	81.4%	77.8%	220b7714
OPENWAYBACK	web wayback machine	66.8k	320	28.0%	26.8%	680fba15
PDFBOX	PDF document manipulation	227.6k	1,587	49.7%	43.3%	d9930344
SCIFIO	scientific image format IO	79.4k	1,019	37.1%	19.3%	281e7ce2
TRACCAR	server for GPS tracking	59.6k	310	56.4%	49.0%	6d259427
URBAN-AIRSHIP	library for marketing platform	37.9k	706	79.3%	46.0%	98edb3ca
VECTORZ	fast vector mathematics	61.9k	456	61.1%	63.8%	a05c69d8

5,254 the most test cases. The line coverage of the projects ranges between 28.0% and 95.0%.

### 6.4.3 Study Design

The research questions were approached as explained below.

**RQ 1:** We hypothesized that the higher the minimal stack distance of a method is to any test case, the more likely the method is to be pseudo-tested. To test this hypothesis, we analyzed whether a correlation exists between a method's minimal stack distance to any test case and its mutation testing verdict (*i.e.*, whether a method is pseudo-tested by *all* test cases or not). For each project, we computed the Spearman's rank correlation coefficient, which expresses the strength of this relationship (between  $-1$  and  $+1$ ), and the p-value, using a significance level of 5%. Moreover, we present plots illustrating the proportion of pseudo-tested methods per minimal stack distance value.

**RQ 2:** To answer this research question, further measures, besides minimal stack distance, were collected of each covered method. The selected measures can all be computed either using a static analysis or alongside a single execution of a test suite. The measures are:

- Line count: number of coverable lines of code in a method
- Branch count: number of branches
- Line coverage: proportion of covered lines out of coverable lines
- Branch coverage: proportion of covered branches out of coverable branches (100% for covered methods without branches)
- Number of covering test cases: number of test cases that execute a method
- Scope of covering test cases: minimum number of covered methods of any of a method's covering test cases (*i.e.*, how many other methods besides the considered method the test cases cover at a minimum)

- Maximum invocation count: maximum number of invocations of a method during the execution of any covering test case
- Return type of a method: void, boolean, numeric, string, array, reference to object

For each project, one machine-learning classifier was trained to predict a method’s mutation testing verdict with respect to all covering test cases, and one was trained to predict the mutation testing verdict of a test-case method pair.

We evaluated the performance of the models with respect to within-project and cross-project predictions. Within-project evaluations show how well predictions work when models are trained on a data-subset of the same project, while cross-project evaluations indicate how well models can be generalized to conduct predictions in other projects. For within-project predictions, repeated 10-fold cross-validation was applied (Kohavi, 1995). For cross-project predictions, each project was tested with a model that is trained on the respective remaining projects.

We measured model performance by computing precision, recall, and F-score. Following Zhang *et al.* (2018), we predicted both outcomes (*pseudo-tested* and *tested*) and used the weighted average of the performance metrics (*i.e.*, “each metric is weighted according to the number of instances with the particular class label”). We also reported the performance of the outcome *pseudo-tested* because methods with this outcome represent the minority class, which makes this outcome more difficult to predict.

Furthermore, we show the prediction model’s variable importances of one project, as an example.

#### 6.4.4 Data Collection and Processing

To collect data for the study, we first executed the test suite of each study object and recorded the minimal stack distance of each test-case method pair. The recording of the stack distance was carried out as defined in Section 6.2

Table 6.2: Example of a full mutation matrix.

Method	Test case	Mutation testing verdict
$m_1$	$t_1$	pseudo-tested
$m_1$	$t_2$	tested
$m_2$	$t_2$	pseudo-tested

and described in Section 6.3. Note that we were working on the existing test suites of the projects; that is, we did not generate test cases.

Second, we conducted a mutation analysis for each study object. For that, we used *PIT* (Coles *et al.*, 2016) in version 1.4.0 with the *pit-mp* extension to support multi-module projects. PIT is a well-known mutation testing tool for Java applications and has been used in several studies (e.g., Ahmed *et al.* (2016), Gopinath *et al.* (2015), and Gopinath *et al.* (2016)). As performance optimization, PIT aborts the analysis of a mutant after the mutant is first killed by a test case. For the present study, however, a full mutation matrix was required, which contains the verdict (killed or survived) of each mutant for each covering test case. Therefore, we adjusted PIT to compute a full mutation matrix, as proposed by Ahmed *et al.* (2016). Table 6.2 presents an example of such a matrix.<sup>1</sup>

PIT was used with the *Descartes* plug-in (Vera-Pérez *et al.*, 2018) in version 1.2.4. This plug-in extends PIT with the mutation operator to uncover pseudo-tested methods (see Section 4.4.2) and uses the return values presented in Table 6.3. We aggregated the analysis results for each method; when two mutants are created, a method is only considered pseudo-tested if both mutants cannot be killed by any test case.

We excluded empty methods and methods solely returning `null` from the analysis because their mutation would result in an equivalent mutant. We also excluded `hashCode` methods because mutation testing is not suitable for assessing their testing state. We further excluded constructors because, as described in the limitations of the stack distance computation in Section 6.3,

<sup>1</sup> Although PIT also reports mutants for uncovered methods, we are only interested in mutants that are covered by at least one test case.



Table 6.3: Return values of the *Descartes* implementation of the operator.

Return type	Mutant 1	Mutant 2
void	<i>(void)</i>	<i>(not created)</i>
boolean	false	true
byte, short, int, long	0	1
float, double	0.0	0.1
char	' '	'A'
string	""	"A"
T[]	new T[]{}	<i>(not created)</i>
reference type	null	<i>(not created)</i>

we could not compute reliable stack distance values of these special methods. Moreover, we excluded generated code, which was present for example in BITCOINJ, because the code is automatically re-generated during the build process and not designed to be tested.

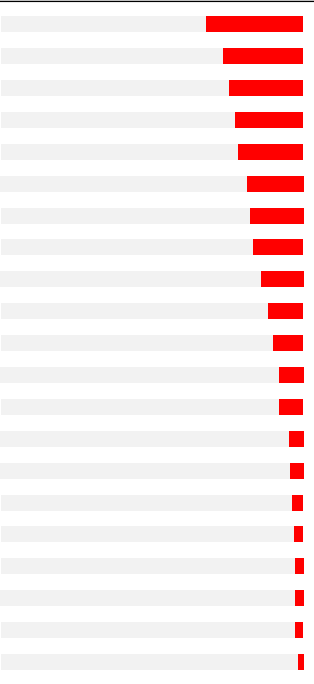
We collected further measures to enhance the prediction model applied in RQ 2. For that, we used *JaCoCo* to compute a method's number of lines and branches, as well as line and branch coverage values. The number of covering test cases per method and their scope was computed based on the full mutation matrix. The method's invocation count during a test execution was collected alongside the stack distance recording. Finally, the return type of a method was deduced from the mutation testing output.

We processed data with the statistical software *R*.<sup>1</sup> We trained and evaluated prediction models with *R*'s *caret* package (Kuhn *et al.*, 2017). *Random Forest* was selected as the machine-learning algorithm because preliminary experiments on our datasets revealed that it achieved the best performance. *adaboost* achieved an almost equal performance, but was approximately 11 times slower. Zhang *et al.* (2018) also used random forest for their predictions.

---

<sup>1</sup> <https://cran.r-project.org>

Table 6.4: Overview of the mutation analysis results.

Project	Pseudo-tested methods out of all <i>mutated</i> methods		
	#		% ↓
SCIFIO	154		32.0%
PDFBOX	829		26.3%
BIOJAVA	1,147		24.4%
TRACCAR	193		22.4%
COMMONS IMAG	244		21.4%
OPENWAYBACK	166		18.6%
JFREECHART	754		17.7%
GOOGLE HTTP	145		16.5%
JAVAPARSER	293		14.0%
GRAPHHOPPER	252		11.5%
GEOMETRY API	224		9.9%
VECTORZ	339		8.0%
JACKSON-DB	307		7.8%
BITCOINJ	77		4.7%
JSOUP	37		4.4%
URBAN-AIRSHIP	78		3.5%
COMMONS GEOM	20		2.8%
GSON	15		2.8%
COMMONS MATH	129		2.7%
COMMONS STAT	7		2.6%
COMMONS LANG	43		1.7%
<i>median</i>	166		9.9%

#### 6.4.5 Results

Before addressing the research questions, Table 6.4 presents the absolute and relative number of pseudo-tested methods of each project, as computed in the mutation analysis. Depending on the project, between 1.7% and 32.0% of the *mutated* methods are pseudo-tested. The proportion is low in GSON and four of the Apache projects, indicating that methods in these projects are tested more effectively than in the other projects. By contrast, the proportion of pseudo-tested methods in SCIFIO, PDFBOX, and BIOJAVA is

Table 6.5: RQ 1: Spearman’s correlation coefficient between a method’s minimal stack distance and its mutation testing verdict. Absolute coefficient values  $\geq 0.2$  and p-values  $< 0.05$  are highlighted.

Project	Coefficient ↓	p-value
JFREECHART	+0.58	<0.001
SCIFIO	+0.48	<0.001
JAVAPARSER	+0.41	<0.001
COMMONS STAT	+0.35	<0.001
TRACCAR	+0.33	<0.001
PDFBOX	+0.31	<0.001
BIOJAVA	+0.29	<0.001
GRAPHHOPPER	+0.24	<0.001
COMMONS LANG	+0.21	<0.001
BITCOINJ	+0.20	<0.001
JACKSON-DB	+0.18	<0.001
JSOUP	+0.18	<0.001
COMMONS GEOM	+0.17	<0.001
COMMONS IMAG	+0.16	<0.001
GEOMETRY API	+0.15	<0.001
OPENWAYBACK	+0.14	<0.001
GSON	+0.13	0.003
URBAN-AIRSHIP	+0.11	<0.001
COMMONS MATH	+0.08	<0.001
VECTORZ	+0.07	<0.001
GOOGLE HTTP	-0.17	<0.001

above average.

**RQ 1: Are methods with a higher minimal stack distance to the test cases more likely to be pseudo-tested?** Table 6.5 shows the results of the Spearman’s correlation test between a method’s minimal stack distance and mutation testing verdict.

We observed that a statistically significant correlation exists in all 21 projects (p-value  $< 0.05$ ). The positive correlation coefficients indicate that the proportion of pseudo-tested methods increases with increasing stack dis-

tance values. The strongest correlation is achieved in the project JFREECHART with a correlation coefficient of 0.58. When looking at this project's test code, it was striking that the test cases contain many assertions. A moderate correlation with a coefficient between 0.3 and 0.5 is present in five further projects. A weak correlation is present in the remaining projects. In the project GOOGLE HTTP a weak negative correlation is observed; however, in this project, the minimal stack distance does not exceed the value 2 in 81% of the methods. The overall correlation based on data from all projects is statistically significant and exhibits +0.26 as coefficient.

The red line in Figure 6.3 presents the proportion of pseudo-tested methods per minimal stack distance value. In the project JFREECHART, more than 50% of the methods with a minimal stack distance higher than 3 are pseudo-tested.

The illustration in Figure 6.4 indicates that the correlation between a method's minimal stack distance and its mutation testing verdict is generally stronger in larger projects with a high proportion of pseudo-tested methods. (The correlation between the project's correlation coefficient and these two project characteristics is 0.4 in each case.)

*Methods with a higher minimal stack distance to covering test cases are more likely to be pseudo-tested.*

**RQ 2: How well can the mutation testing verdict be predicted using test-relationship measures?** Table 6.6 presents the classifier's precision, recall, and F-score of the within-project prediction of a method's mutation testing verdict. As described in Section 6.4.3, the performance measures constitute the weighted average of the outcomes *pseudo-tested* and *tested*. Median precision is 92.9%, and median recall is 93.4%. When conducting cross-project prediction at the same level, median precision and recall deteriorate to 85.6% and 88.1%, respectively.

Pseudo-tested methods represent the minority class and are therefore more difficult to predict. Table 6.7 shows the within-project prediction

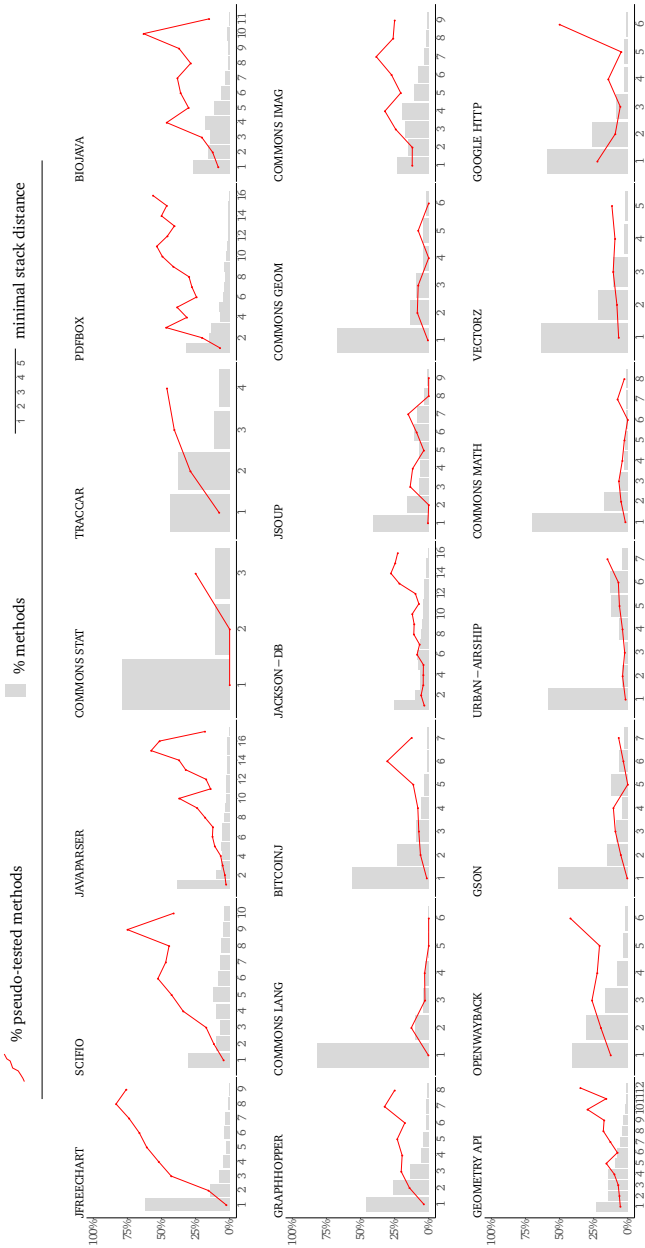


Figure 6.3: RQ 1: **—** Proportion of pseudo-tested methods (red line) and **■** proportion of methods per minimal stack distance value (gray bars). The hypothesis is that the proportion of pseudo-tested methods increases with increasing minimal stack distance values. The x-axis is cropped when the proportion of methods per distance value falls below 0.5%.

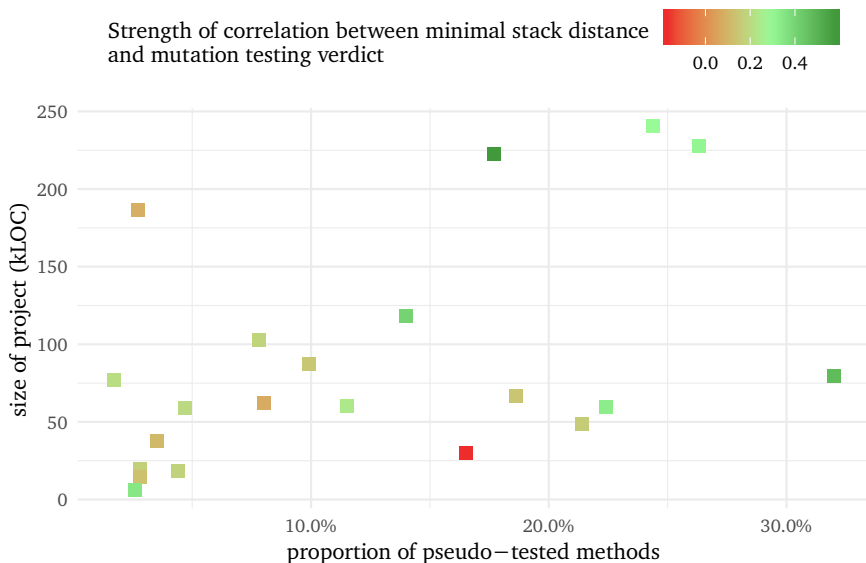


Figure 6.4: The projects’ proportion of pseudo-tested methods (x-axis), project size in kLOC (y-axis), and the strength of the correlation between a method’s minimal stack distance and its mutation testing verdict from Table 6.5 (color).

performance for identifying pseudo-tested methods. Median precision of this outcome is 70.7% and median recall is 34.3%. In the best case, 96.6% precision and 100% recall are still achieved (COMMONS STAT).

Figure 6.5 presents the variable importance of JFREECHART’s within-project prediction model. The figure shows that the minimal stack distance and the minimal scope value of a method’s covering test cases (the scope of a test case expresses how many methods it covers) are the most important variables for the prediction model.

Cross-project prediction for identifying pseudo-tested methods achieves poor performance. Even when additionally pre-processing training sets with *synthetic minority over-sampling technique (SMOTE)* (Chawla et al., 2002), median precision is only 19.2% and median recall is 43.2%. Hence, cross-project prediction is not well suited to uncovering pseudo-tested methods.

Table 6.6: RQ 2: Performance when predicting a *method's mutation testing verdict*.

Project	Precision	Recall	F-score ↓
COMMONS STAT	99.9%	99.9%	99.9%
COMMONS LANG	98.8%	98.9%	98.7%
GSON	97.5%	97.7%	97.1%
COMMONS MATH	96.7%	97.5%	96.7%
COMMONS GEOM	96.2%	97.2%	96.4%
URBAN-AIRSHIP	96.3%	96.9%	96.3%
GOOGLE HTTP	95.1%	95.1%	94.9%
JSOUP	94.1%	95.6%	94.3%
BITCOINJ	93.7%	95.3%	94.0%
JFREECHART	93.1%	93.4%	93.1%
JAVAPARSER	92.9%	93.2%	92.8%
VECTORZ	92.5%	93.5%	92.4%
JACKSON-DB	91.5%	93.0%	91.7%
GRAPHHOPPER	89.5%	90.8%	89.3%
GEOMETRY API	86.6%	90.0%	87.1%
TRACCAR	86.8%	87.1%	86.9%
COMMONS IMAG	87.2%	87.7%	86.8%
BIOJAVA	85.1%	85.7%	85.1%
PDFBOX	84.1%	84.7%	83.8%
OPENWAYBACK	81.3%	83.5%	81.4%
SCIFIO	78.7%	79.0%	78.8%
<i>median</i>	<i>92.9%</i>	<i>93.4%</i>	<i>92.8%</i>

*The mutation testing verdict of a method can on average be predicted with 92.9% precision and 93.4% recall. Cross-project prediction is more challenging and achieves weaker performance.*

The above results concern the prediction of a method's mutation testing verdict with respect to all test cases. For other use cases, such as enhancing test case prioritization with test effectiveness information, it can also be useful to predict the mutation testing verdict of a test-case method *pair*.

Table 6.7: RQ 2: Performance when predicting *pseudo-tested methods*.

Project	Precision	Recall	F-score ↓
COMMONS STAT	96.6%	100.0%	98.2%
GOOGLE HTTP	94.6%	74.8%	83.5%
JFREECHART	87.0%	73.4%	79.6%
JAVAPARSER	84.1%	63.4%	72.3%
TRACCAR	72.6%	68.0%	70.2%
BIOJAVA	76.4%	59.9%	67.1%
PDFBOX	78.4%	57.6%	66.4%
SCIFIO	68.5%	63.8%	66.1%
COMMONS IMAG	81.1%	55.5%	65.9%
COMMONS LANG	85.5%	41.3%	55.7%
GRAPHHOPPER	70.6%	34.3%	46.2%
VECTORZ	70.7%	32.5%	44.5%
OPENWAYBACK	60.7%	32.5%	42.4%
URBAN-AIRSHIP	64.2%	27.6%	38.6%
JACKSON-DB	61.4%	26.6%	37.1%
GSON	87.5%	23.3%	36.8%
COMMONS MATH	60.3%	15.9%	25.2%
COMMONS GEOM	50.0%	15.0%	23.1%
BITCOINJ	50.0%	13.0%	20.6%
JSOUP	51.5%	11.5%	18.8%
GEOMETRY API	46.9%	11.0%	17.9%
<i>median</i>	<i>70.7%</i>	<i>34.3%</i>	<i>46.2%</i>

Table 6.8 presents the within-project performance when predicting the mutation testing verdict of a test-case method pair. At this level, median precision and recall are 84.8% and 85.3%, respectively. When focusing on the outcome *pseudo-tested*, median precision and recall still achieve 82.4% and 71.7%, respectively.

Hence, the prediction achieves promising results when working on test-case method pairs. One reason for this is that, unlike when predicting the mutation testing verdict of a method with respect to all test cases, test case metrics are not aggregated.



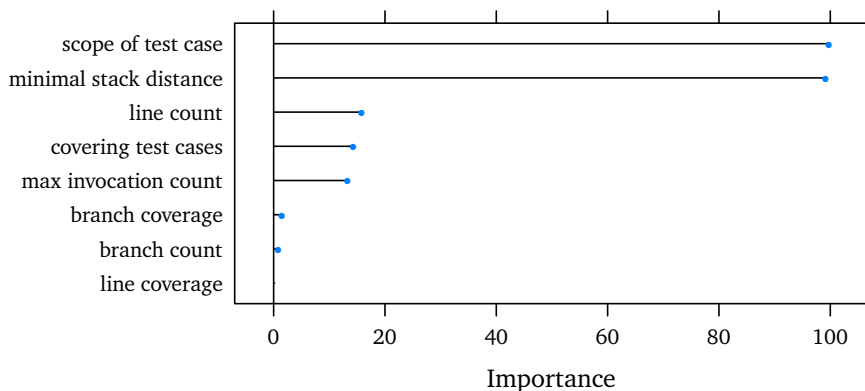


Figure 6.5: RQ 2: Variable importance of JFREECHART’s prediction model (scaled to one).

*Pseudo-tested test-case method pairs can be predicted with 82.4% precision and 71.7% recall, on average.*

Zhang *et al.* (2018) achieved precision and recall values of around 90% (depending on project and scenario). They only presented performance measures aggregated of both outcomes. Although an in-depth comparison with their results does not seem sensible—because they predicted for different mutation operators, used other metrics, and included methods not covered by any test case—we can still say that the prediction performance is roughly comparable.

#### 6.4.6 Threats to Validity

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

*Threats to internal validity:* The computation of the stack distance is a threat to internal validity. Although great care was taken when developing the computation logic, the implementation could contain faults that affect the outcome. To mitigate this threat, we verified computed values of different

Table 6.8: RQ 2: Performance when predicting the mutation testing verdict of a test-case method pair.

Project	Precision	Recall	F-score ↓
SCIFIO	92.8%	92.8%	92.8%
COMMONS STAT	92.1%	92.4%	91.7%
COMMONS GEOM	90.8%	91.2%	90.4%
JAVAPARSER	90.1%	90.2%	90.1%
URBAN-AIRSHIP	89.1%	90.2%	88.7%
GOOGLE HTTP	88.4%	88.6%	88.2%
GSON	87.9%	88.0%	87.9%
COMMONS LANG	87.5%	87.8%	86.8%
JFREECHART	86.5%	86.5%	86.4%
BITCOINJ	86.1%	86.1%	86.1%
COMMONS MATH	85.2%	85.7%	85.1%
TRACCAR	85.1%	85.0%	85.1%
VECTORZ	85.4%	86.6%	84.9%
JSOUP	84.4%	84.9%	84.1%
PDFBOX	83.8%	83.8%	83.8%
COMMONS IMAG	82.5%	82.7%	82.2%
BIOJAVA	81.7%	81.7%	81.5%
OPENWAYBACK	80.9%	80.8%	80.8%
GRAPHHOPPER	80.6%	80.7%	80.5%
GEOMETRY API	77.6%	78.1%	77.4%
JACKSON-DB	72.4%	72.4%	72.4%
<i>median</i>	<i>85.4%</i>	<i>86.1%</i>	<i>85.1%</i>

code samples and developed automated tests to check the implementation. In addition, the source code of our tool<sup>1</sup> can be inspected on GitHub.

The same applies to the conducted extension of the PIT mutation testing tool to enable computing a full mutation matrix. To mitigate this threat, we created a pull request (Niedermayr, 2018), which was carefully reviewed and merged by the head developer of PIT.

Some of the generated mutants may be equivalent mutants, which differ

<sup>1</sup> <https://github.com/cqse/test-analyzer>

syntactically but not semantically from the original source code, and, therefore cannot be killed (Grün *et al.*, 2009). Consequently, some of the methods that were identified to be pseudo-tested could be equivalent mutants and affect the results. Due to the design of the mutation operator (see Section 4.3) and the exclusion of empty methods and methods returning null, the number of equivalent mutants is expected to be negligible (Niedermayr *et al.*, 2016). A manual review on a sample of 30 pseudo-tested methods confirmed this observation.

*Threats to external validity:* Although we selected 21 study objects with different characteristics, the selection of the projects poses a threat to external validity. Since we chose only open-source projects that use Maven as build system and in which nearly all tests succeed, the sample may contain an over-representation of well-engineered projects with mature test suites. Therefore, future work is necessary in order to determine whether the results are generalizable for Java projects and projects in other programming languages.

## 6.5 Discussion

The study's results show that the correlation between a method's minimal stack distance and its mutation testing verdict is moderate to strong in six projects, and also present in further projects to lower degrees. The correlation is generally stronger in larger projects (JFREECHART, BIOJAVA, PDFBOX), which also exhibit higher minimal stack distance values. In large, multi-module projects, some methods are only tested by integration tests, which usually have a higher distance to many of the covered methods than a unit test has. In such projects, the minimal stack distance can provide valuable insights into the testing state of methods, thereby providing additional value to coverage information.

The evaluation of the prediction models shows that machine-learning models can successfully predict the mutation testing verdict of a method. Hence, such models can be considered as a light-weight alternative to a mutation

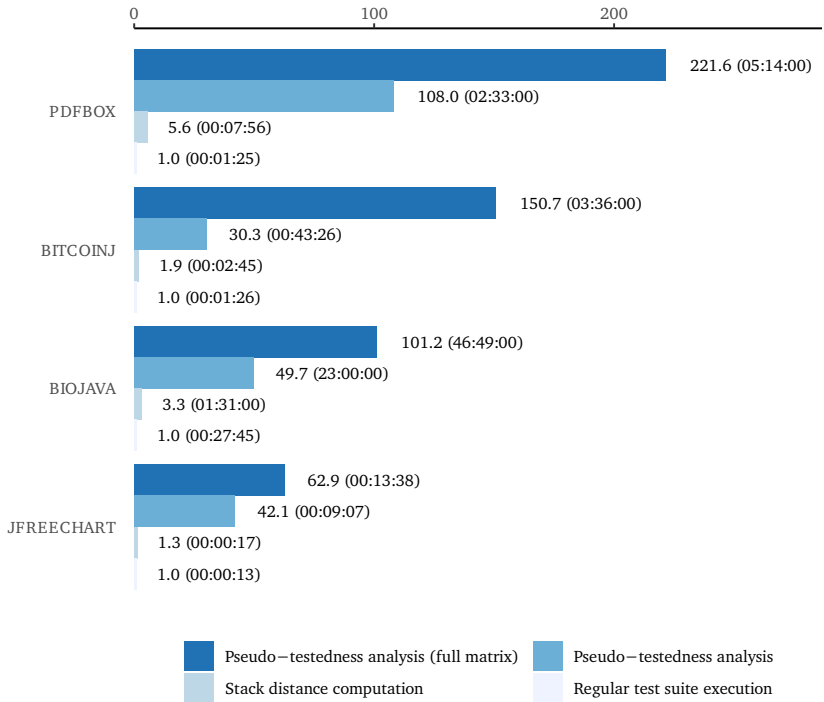


Figure 6.6: Duration of analyses in hours and slowdown factor based on the normal test suite execution.

analysis. To point out possible time savings, Figure 6.6 shows the duration of different analyses exemplarily of four projects. The current—not yet performance-optimized—implementation for recording the minimal stack distance slows the test execution down by a single-digit factor. Nonetheless, a prediction model using this metric can achieve significant savings compared to the execution of a mutation analysis. The analysis with the state-of-the-art mutation testing tool PIT takes approximately 50–200 times

as long as a single execution of the corresponding test suite. In the project BIOJAVA, an analysis with the pseudo-testedness mutation operator that stops assessing a mutant after encountering the first killing test case took 23 hours (approx. 50 times the duration of the test suite execution), and an analysis to compute a full mutation matrix took more than 46 hours (101 times the duration of the test execution). Consequently, prediction models can be an alternative for projects in which a mutation analysis is not applicable due to a long duration.

## 6.6 Summary

We proposed and studied the minimal stack distance measure, which describes the proximity of a method to any of its test cases. The results indicate that a correlation exists between this measure and a method's mutation testing verdict. Hence, we suggest that the minimal stack distance measure can be a useful indicator for determining whether a method is pseudo-tested. Classifiers that predict a method's mutation testing verdict achieve a median precision of 92.9% and recall of 93.4%. The measures needed for such a classifier can be computed in a single test suite execution. By contrast, mutation testing takes by orders of magnitude longer, often resulting in an analysis duration of several hours, days, or weeks, depending on the size of an application. Therefore, we suggest considering such classifiers as a light-weight alternative to mutation testing or as a preceding, less costly step. In particular, the classifiers can be a reasonable alternative in continuous integration. Furthermore, they can be useful for projects in which a mutation analysis is not applicable, which can be due to the analysis duration or other issues (*e.g.*, the use of special class loaders or special dependency-injection mechanisms).



CHAPTER  
7

# IDENTIFYING LOW-FAULT-RISK METHODS

*This chapter describes how methods can be identified that do not necessarily need to be tested due to a low fault risk. These methods can be omitted from a mutation analysis to make it even faster and its results more relevant.*

## 7.1 Motivation

In a perfect world, it would be possible to completely test every new version of a software application before it was deployed into production. In practice, however, software development teams often face a problem of scarce test resources. Developers are busy implementing features and bugfixes, and may lack time to develop enough automated tests to comprehensively test new code (Menzies and Di Stefano, 2004; Ostrand *et al.*, 2005). Furthermore, testing is costly and, depending on the criticality of a system, it may not be cost-effective to expend equal test effort to all components (Zhang *et al.*, 2007). Hence, to cope with the problem of scarce test resources, development teams need to prioritize and limit their testing scope by restricting the code regions to be tested (Bertolino, 2007; Menzies *et al.*, 2003); that is, development teams aim to focus testing on code regions that have the best cost-benefit ratio regarding fault detection. To support development teams in this activity, defect prediction has been developed and studied extensively in the last decades (Catal, 2011; D'Ambros *et al.*, 2012; Hall *et al.*, 2012). Defect prediction identifies code regions that are likely to contain a fault and should therefore be tested (Menzies *et al.*, 2007; Weyuker and Ostrand, 2008).

For example, methods vary in the complexity of their code. Some methods contain a lot of code with a complex control flow and deep nesting. Other methods are short and just do a simple task, such as, delegating a method call or returning a stored value. Hence, it is evident that not all methods share the same fault risk and not all methods justify the same testing efforts.

Although recent research in defect prediction shows progress in cross-project predictions (Xia *et al.*, 2016; Xu *et al.*, 2018; Zhang *et al.*, 2016), defect-prediction models often face the problem that they are only applicable to the project on which the model was trained; that is, they are not generalizable to other projects (He *et al.*, 2012; Turhan *et al.*, 2009; Zimmermann *et al.*, 2009). This is a problem because precise data to train such models is difficult to obtain (Shippey *et al.*, 2016).

In this chapter, we suggest, implement, and evaluate another view on



defect prediction: inverse defect prediction (IDP). The idea behind IDP is to identify code artifacts (*e.g.*, methods) that are very unlikely to exhibit any faults due to their source code's characteristics so that they can be deferred or ignored in testing. Hence, instead of identifying methods that are likely to contain a fault, we focus on methods with *trivial* code, which may be less project-specific than complex methods, allowing the models to be generalizable. We investigate whether such low-fault-risk methods exist and study whether IDP can reliably identify these methods. Our goal is not to predict all methods that do not contain any faults in the training set; we want to optimize for precision and classify a method as “low fault risk” only if we are very certain. To implement IDP, we calculated code metrics for each method of a code base and trained a classifier for methods with low fault risk using association rule mining. To evaluate IDP, we performed an empirical study with the Defects4J dataset (Just *et al.*, 2014b) consisting of real faults from six open-source projects. We applied static code analysis and classifier learning on these code bases and evaluated the performance in with-in project and cross-project prediction scenarios. Our results show that IDP can be used to identify low-fault-risk methods and is also well applicable in cross-project predictions. We suggest that low-fault-risk methods would be *too trivial to test* and conclude that they constitute only a very low risk when they are pseudo-tested. Hence, the identified methods can be excluded from a mutation analysis to allow focusing on the more relevant methods and to further reduce the analysis duration. Besides that, the identification of such low-fault-risk method can support a better allocation of quality-assurance resources, for example, when prioritizing testing efforts for changed code that is not covered by any test cases yet.

*The main contributions of this chapter are:*

- We present the idea of an inverse view on defect prediction. While defect prediction has been studied extensively in the last decades, it has always been employed to identify code regions with *high* fault risk. To the best of our knowledge, this is the first study explicitly targeting

the identification of code regions with *low* fault risk.

- We present an empirical study to evaluate the performance of IDP on open-source projects with real faults.
- We provide an extension to the Defects4J dataset (Just *et al.*, 2014b). To improve data quality and enable further research—reproduction in particular—we provide code metrics for all methods in the code bases and an indication whether they were changed in a bugfix patch, a list of methods that changed in bugfixes only to preserve API compatibility, and association rules to identify low-fault-risk methods.

Parts of the content of this chapter have been published in Niedermayr *et al.* (2018b) and Niedermayr *et al.* (2019).

## 7.2 Approach

This section describes the inverse defect prediction approach, which identifies low-fault-risk methods. The approach comprises the computation of source-code metrics for each method, the data pre-processing before the mining, and the association rule mining. Figure 7.1 illustrates the steps.

### 7.2.1 Metric Computation

Like defect prediction models, IDP uses metrics to train a classifier for identifying low-fault-risk methods. For each method, we compute the source-code metrics listed in Table 7.1 that we considered relevant to judge whether a method is trivial. They comprise established length and complexity metrics used in defect prediction, metrics regarding occurrences of programming-language constructs, and categories describing the purpose of a method.

*Source lines of code (SLOC)* is a metric to measure the number of source lines of code; that is, LOC without empty lines and comments. *Cyclomatic Complexity* corresponds to the metric proposed by McCabe (1976). Despite this metric being controversial (Hummel, 2014; Shepperd, 1988)—due

Table 7.1: Computed metrics for each method.

	Metric name	Type
M1	Source Lines of Code (SLOC)	length
M2	Cyclomatic Complexity	complexity
M3	Max. Nesting Depth	max. value
M4	Max. Method Chaining	max. value
M5	Unique Variable Identifiers	unique count
M6	Anonymous Class Declarations	count
M7	Arithmetic In- or Decrementations	count
M8	Arithmetic Infix Operations	count
M9	Array Accesses	count
M10	Array Creations	count
M11	Assignments	count
M12	Boolean Operators	count
M13	Cast Expressions	count
M14	Catch Clauses	count
M15	Comparison Operators	count
M16	If Conditions	count
M17	Inner Method Declarations	count
M18	Instance-of Checks	count
M19	Instantiations	count
M20	Loops	count
M21	Method Invocations	count
M22	Null Checks	count
M23	Null Literals	count
M24	Return Statements	count
M25	String Literals	count
M26	Super-Method Invocations	count
M27	Switch-Case Blocks	count
M28	Synchronized Blocks	count
M29	Ternary Operations	count
M30	Throw Statements	count
M31	Try Blocks	count
M32	All Conditions	count
M33	All Arithmetic Operations	count
M34	Is Constructor	boolean
M35	Is Setter	boolean
M36	Is Getter	boolean
M37	Is Empty Method	boolean
M38	Is Delegation Method	boolean
M39	Is ToString Method	boolean

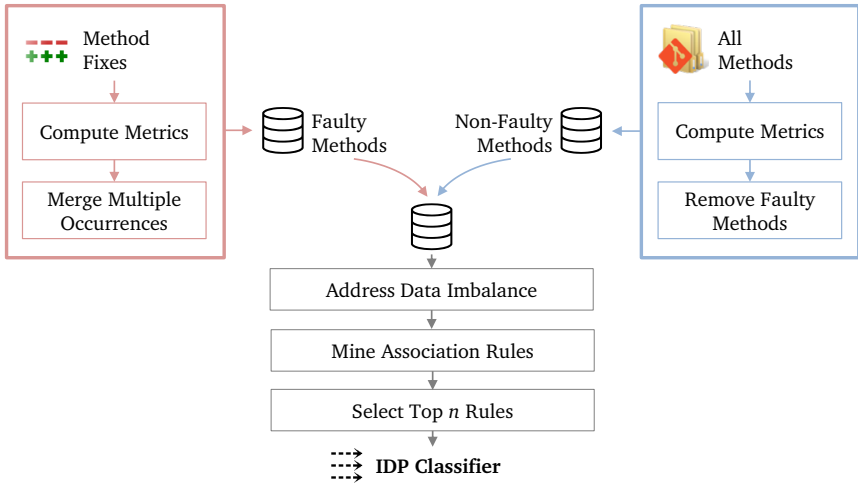


Figure 7.1: Overview of the approach to identify low-fault-risk methods. Metrics of faulty methods are computed at the faulty state; metrics of non-faulty methods are computed at the state of the last bugfix commit.

to the fact that it is not actionable, difficult to interpret, and high values do not necessarily translate to low readability—it is commonly used as a variable in defect prediction (Menzies *et al.*, 2002; Menzies *et al.*, 2004; Zimmermann *et al.*, 2007). Furthermore, a low number of paths through a method could be relevant for identifying low-fault-risk methods. *Maximum Nesting Depth* corresponds to the “maximum number of encapsulated scopes inside the body of the method” (ndepend, 2017). Deeply nested code is more difficult to understand, therefore, it could be more fault-prone. *Maximum Method Chaining* expresses the maximum number of chain elements of a method invocation. We consider a method call to be chained if it is directly invoked on the result from the previous method invocation. The value for a method is zero if it does not contain any method invocations, one if no method invocation is chained, or otherwise the maximum number of chain elements (e.g., two for `getId().toString()`, three for

`getId().toString().substring(1)`). *Unique Variable Identifiers* counts the distinct names of variables that are used within a method. The following metrics, M6 to M31, count the occurrences of the respective Java language construct (Gosling *et al.*, 2013).

Next, we derive further metrics from the existing ones. They are redundant, but correlated metrics do not have any negative effects on association rule mining (except on the computation time) and may improve the results for the following reason: if an item generated from a metric is not frequent, rules with this item will be discarded because they cannot achieve the minimum support; however, an item for a more general metric may be more frequent and survive. The derived metrics are:

- *All Conditions*, which sums up *If Conditions*, *Switch-Case Blocks*, and *Ternary Operations* (M16 + M27 + M29)
- *All Arithmetic Operations*, which sums up *Incrementations*, *Decrementations*, and *Arithmetic Infix Operations* (M7 + M8)

Furthermore, we compute to which of the following categories a method belongs (a method can belong to zero, one, or more categories):

- *Constructors*: Special methods that create and initialize an instance of a class. They might be less fault-prone because they often only set class variables or delegate to another constructor.
- *Getters*: Methods that return a class or an instance variable. They usually consist of a single statement and can be generated by the IDE.
- *Setters*: Methods that set the value of a class or an instance variable. They usually consist of a single statement and can be generated by the IDE.
- *Empty Methods*: Non-abstract methods without any statements. They often exist to meet an implemented interface, or because the default logic is to do nothing and is supposed to be overridden in certain sub-classes.

- *Delegation Methods*: Methods that delegate the call to another method with the same name and further parameters. They often do not contain any logic besides the delegation.
- *ToString Methods*: Implementations of Java's `toString` method. They are often used only for debugging purposes and can be generated by the IDE.

Note that we only use source-code metrics and do not consider process metrics. This is because we want to identify methods that exhibit a low fault risk due to their *code*.

Association rule mining computes frequent itemsets from categorical attributes; therefore, our next step is to discretize the numerical metrics. In defect prediction, discretization is also sometimes applied to the metrics: [Shivaji et al. \(2013\)](#) and [McCallum and Nigam \(1998\)](#) reported that binary values can yield better results than counters when the number of features is low. We discretize as follows:

- For each of the metrics M1 to M5, we inspect their distribution and create three classes. The first class is for metric values until the first tertile, the second class for values until the second tertile, and the third class for the remaining values.
- For all count metrics (including the derived ones), we create a binary “has-no”-metric, which is true if the numerical value is zero, for example,  $CountLoops = 0 \Rightarrow NoLoops = true$ .
- For the method categories (setter, getter, ...), no transformation is necessary because they are already binary.

### 7.2.2 Data Pre-Processing

At this point, we assume that we have a list of faulty methods with their metrics at the faulty state (the list may contain a method multiple times if it was fixed multiple times) and a list of all methods. Faulty methods can be obtained by identifying methods that were changed in bugfix commits ([Giger](#)

*et al.*, 2012; Shippey *et al.*, 2016; Zimmermann *et al.*, 2007). A method is considered as faulty when it was *faulty* at least once in its history; otherwise it is considered as *not faulty*. We describe in Section 7.3.3 how we extracted faulty methods from the Defects4J dataset.

Prior to applying the mining algorithm, we have (1) to address faulty methods with multiple occurrences, (2) to create a unified list of faulty and non-faulty methods, and (3) to tackle dataset imbalance.

Steps (1) and (2) require that a method can be uniquely identified. To satisfy this requirement, we identified a method by its name, its parameter types, and the qualified name of its surrounding class. We integrated the computation of the metrics into the software-quality analysis suite tool *Teamscale* (Haas *et al.*, 2019c), which is aware of the code history and tracks method genealogies. Thereby, *Teamscale* detects method renames or parameter changes so that we could update the method identifier when it changed.

- (1) A method may be fixed multiple times; in this case, a method appears multiple times in the list of the faulty methods. However, each method should have the same weight and should therefore be considered only once. Consequently, we consolidate multiple occurrences of the same method: we replace all occurrences by a new instance and apply majority voting to aggregate the binary metric values. It is common practice in defect prediction to have a single instance of every method with a flag, which indicates whether a method was faulty at least once (Giger *et al.*, 2012; Mende and Koschke, 2009; Menzies *et al.*, 2010; Shippey *et al.*, 2016).
- (2) To create a unified dataset, we take the list of all methods, remove methods that exist in the set of the faulty methods, and add the set of the faulty methods with the metrics computed *at the faulty state*. After doing that, we end up with a list containing each method exactly once and a flag indicating whether a method was faulty or not.
- (3) Defect datasets are often highly imbalanced (Khoshgoftaar *et al.*, 2010), with faulty methods being underrepresented. Therefore, we apply

SMOTE, a well-known algorithm for over- and under-sampling, to address imbalance in the dataset used for training (Chawla *et al.*, 2002; Longadge *et al.*, 2013). It artificially generates new entries of the minority class using the nearest neighbors of these cases and reduces entries from the majority class (Torgo, 2010). If we do not apply *SMOTE* to highly imbalanced datasets, many non-expressive rules will be generated when most methods are not faulty. For example, if 95% of the methods are not faulty and 90% of them contain a method invocation, rules with high support will be generated that use this association to identify non-faulty methods. Balancing avoids those nonsense rules.

### 7.2.3 IDP Classifier

To identify low-fault-risk methods, we compute association rules of the type  $\{ \textit{Metric1}, \textit{Metric2}, \textit{Metric3}, \dots \} \rightarrow \{ \textit{NotFaulty} \}$ . Examples of the metrics are *SlocLowestThird*, *NoNullChecks*, *IsSetter*. A method that satisfies all metric predicates of a rule is not faulty to the certainty expressed by the confidence of the rule. The support of a rule expresses how many methods with these characteristics exist, and thus, it shows how generalizable the rule is.

After computing the rules on a training set, we remove redundant ones (see the background on association rule mining in Section 2.5.2.2) and order the remaining rules first descending by their confidence and then by their support. To build a low-fault-risk classifier, we combine the top  $n$  association rules with the highest confidence values using the logical-or operator. Hence, we consider a method as having a low fault risk if at least one of the top  $n$  rules matches. To determine  $n$ , we compute the maximum number of rules until the faulty methods in the low-fault-risk methods exceed a certain threshold in the training set.

Of course, IDP can also be used with other machine-learning algorithms. We decided to use association rule mining because of the natural comprehensibility of the rules (see Section 2.5.2.2) and because we achieved better performance compared to models we trained using random forest.



## 7.3 Study

In this study, we researched how well methods that contain hardly any faults can be identified and whether IDP is applicable in cross-project scenarios. We further compared the performance of IDP with a traditional defect prediction approach.

### 7.3.1 Research Questions

We investigated the following research questions.

**RQ 1: What is the precision of the classifier for low-fault-risk methods?** To evaluate the precision of the IDP classifier, we investigated how many methods that are classified as “low fault risk” (due to the triviality of their code) are faulty. If we wanted to use the low-fault-risk classifier for determining methods that require less focus during QA activities (such as testing and code reviews), we would need to be sure that these methods contain hardly any faults.

**RQ 2: How large is the fraction of the code base consisting of methods classified as “low fault risk”?** We studied how common low-fault-risk methods are in code bases to find out how much code is of lower importance for QA activities. We wanted to determine which savings potential can arise when these methods are excluded from QA.

**RQ 3: Is a trained classifier for methods with low fault risk generalizable to other projects?** Cross-project defect prediction is used to predict faults in (new) projects, for which no historical fault data exists, by using models trained on other projects. It is considered a challenging task in defect prediction (He *et al.*, 2012; Turhan *et al.*, 2009; Zimmermann *et al.*, 2009). As we expected that the characteristics of low-fault-risk methods might be project-independent, IDP could be well applicable in a cross-project scenario. Therefore, we investigated how generalizable our IDP classifier is for cross-project use.

**RQ 4: How does the classifier perform compared to a traditional defect prediction approach?** The main purpose of defect prediction is to detect fault-prone code. Most traditional defect prediction approaches are binary classifications, which classify a method either as (likely) faulty or not faulty. Hence, they implicitly also detect methods with a low fault risk. Therefore, we wanted to compare the performance of our classifier with the performance of a traditional defect prediction approach.

### 7.3.2 Study Objects

For our analysis, we used data from Defects4J (Just *et al.*, 2014b), which is a database and analysis framework that provides real faults from six real-world open-source projects written in Java. For each fault, the original commit before the bugfix (faulty version), the original commit after the bugfix (fixed version), and a minimal patch of the bugfix are provided. The patch is minimal such that it contains only code changes that (1) fix the fault and (2) are necessary to keep the code compilable (*e.g.*, when a bugfix involves method-signature changes). It does not contain changes that do not influence the semantics (*e.g.*, changes in comments, local renamings), and changes that were included in the bugfix commit, but are not related to the actual fault (*e.g.*, refactorings). Due to the manual analysis, this dataset at the method level is much more precise than other datasets at the same level, such as Shippey *et al.* (2016) and Giger *et al.* (2012), which were generated from version control systems and issue trackers without further manual filtering. The authors of Just *et al.* (2014b) confirmed that they considered every bugfix within a given time span.

Table 7.2 presents the study objects and their characteristics. We computed the metrics *SLOC* and *#Methods* for the code revision at the last bugfix commit of each project; the numbers do not comprise sample and test code. *#Faulty methods* corresponds to the number of faulty methods derived from the dataset.

Table 7.2: Study objects from the Defects4J data set.

Name	SLOC	# Methods	# Faulty methods
JFREECHART (CHART)	81.6 k	6.8 k	39
GOOGLE CLOSURE COMPILER	166.7 k	13.0 k	148
APACHE COMMONS LANG	16.6 k	2.0 k	73
APACHE COMMONS MATH	9.5 k	1.2 k	132
MOCKITO	28.3 k	2.5 k	64
JODA TIME	89.0 k	10.1 k	45

### 7.3.3 Fault Data Extraction

Defects4J provides for each project a set of reverse patches,<sup>1</sup> which represent bugfixes. To obtain the list of methods that were at least once faulty, we conducted the following steps for each patch. First, we checked out the source code from the project repository at the original bugfix commit and stored it as *fixed version*. Second, we applied the reverse patch to the fixed version to get to the code before the bugfix and stored the resulting *faulty version*.

Next, we analyzed the two versions created for every patch. For each file that was changed between the faulty and the fixed version, we parsed the source code to identify the methods. We then mapped the code changes to the methods to determine which methods were touched in the bugfix. After that, we had the list of faulty methods. Figure 7.2 summarizes these steps.

We inspected all 395 bugfix patches and found that 10 method changes in the patches do not represent bugfixes. While the patches are minimal such that they contain only bug-related changes (see Section 7.3.2), these 10 method changes are semantic-preserving, only necessary because of changed signatures of other methods in the patch, and therefore included in Defects4J to keep the code compilable. Figure 7.3 presents an example. Although these methods are part of the bugfix, they were not changed semantically and do not represent faulty methods. Therefore, we decided to remove them

<sup>1</sup> A reverse patch reverts previous changes.

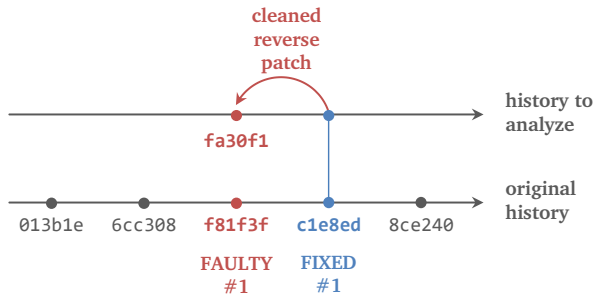


Figure 7.2: Derivation of faulty methods. The original bugfix commit `c1e8ed` to fix the faulty version `f81f3f` may contain unrelated changes. Defect4J provides a reverse patch, which contains only the actual fix. We applied it to the fixed version `c1e8ed` to get to `fa30f1`. We then identified methods that were touched by the patch and computed their metrics at state `fa30f1`.

from the faulty methods in our analysis. The names of these 10 methods are provided in [Niedermayr et al. \(2018a\)](#).

### 7.3.4 Study Design

This section describes the execution of IDP and how we approached the research questions.

After extracting the faulty methods from the dataset, we computed the metrics listed in Section 7.2.1. We computed them for all faulty methods at their faulty version and for all methods of the application code<sup>1</sup> at the state of the fixed version of the last patch. We used Eclipse JDT AST<sup>2</sup> to create an AST visitor for computing the metrics. For all further processing, we used the statistical software R.<sup>3</sup>

To discretize the metrics M1 to M5, we first computed their value distribution. Figure 7.4 shows that their values are not normally distributed (most

<sup>1</sup> code without sample and test code

<sup>2</sup> <http://www.eclipse.org/jdt>

<sup>3</sup> <https://cran.r-project.org>

```

... */
public static String escapeJavaScript(String str) {
+ return escapeJavaStyleString(str, true, true);
- return escapeJavaStyleString(str, true);
}
}

@@ -152,12 +152,12 @@ public class StringEscapeUtils {
... */
private static String escapeJavaStyleString(String str,
+ boolean escapeSingleQuotes, boolean escapeForwardSlash) {
- boolean escapeSingleQuotes) {
... if (str == null) {

```

Figure 7.3: Example of a method change without behavior modification to address API changes. `escapeJavaScript(String)` invokes `escapeJavaStyleString(String, boolean, boolean)`. A further parameter was added to the invoked method; therefore, it was necessary to adjust the invocation in `escapeJavaScript(String)`. For invocations with the parameter value `true`, the behavior does not change [LANG, patch 46, simplified].

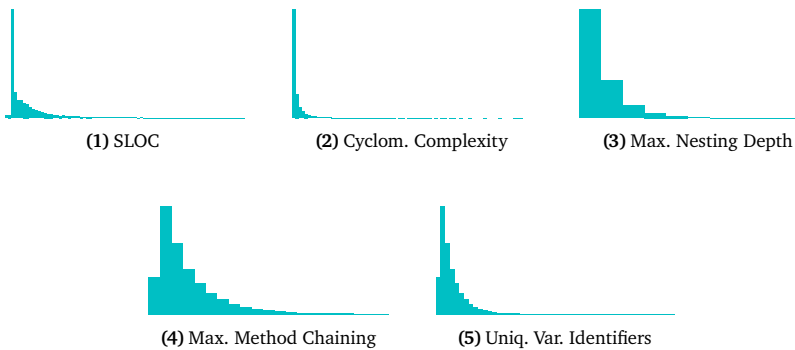


Figure 7.4: Metrics M1 to M5 are not normally distributed. (1) SLOC, (2) Cyclomatic Complexity, (3) Maximum Nesting Depth, (4) Maximum Method Chaining, and (5) Unique Variable Identifiers.

Table 7.3: Generated classes and their value ranges.

Metric	Class 1	Class 2	Class 3
SLOC	[0; 3]	[4; 8]	[9; $\infty$ )
Cyclomatic Complexity	[1; 1]	[2; 2]	[3; $\infty$ )
Maximum Nesting Depth	[0; 0]	[1; 1]	[2; $\infty$ )
Maximum Method Chaining	[0; 1]	[2; 2]	[3; $\infty$ )
Unique Variable Identifiers	[0; 1]	[2; 3]	[4; $\infty$ )

values are very small). To create three classes for each of these metrics, we sorted the metric values, and computed the values at the end of the first and at the end of the second third.<sup>1</sup> We then put all methods until the last occurrence of the highest value of the first third into class 1, all methods until the last occurrence of the highest value of the second third into class 2, and all other methods into class 3. Table 7.3 presents the value ranges of the resulting classes. The classes are the same for all six study objects.

We then aggregated multiple faulty occurrences of the same method (this occurs if a method is changed in more than one bugfix patch) and created a unified dataset of faulty and non-faulty methods (see Section 7.2.2).

Next, we split the dataset into a training and a test set. For RQ 1 and RQ 2, we used 10-fold cross-validation (Witten *et al.*, 2016). Using the *caret* package (Kuhn *et al.*, 2017), we randomly sampled the dataset of each project into 10 stratified partitions of equal sizes. Each partition is used once for testing the classifier, which is trained on the remaining nine partitions. To compute the association rules for RQ 3—in which we study how generalizable the classifier is—for each project, we used the methods of the other five projects as training set for the classifier.

Before computing association rules, we applied the SMOTE algorithm from the *DMwR* package (Torgo, 2010) with a 100% over-sampling and a 200% under-sampling rate to each training set. After that, each training set

<sup>1</sup> We did not use R's *ntile* function to create classes because it always generates classes of the same size such that instances with the same value may end up in different classes (e.g., if 50% of the methods have the complexity value 1, the first 33.3% will end up in class 1, and the remaining 16.7% with the same value will end up in class 2).

was equally balanced (50% faulty methods, 50% non-faulty methods).<sup>1</sup>

We then used the implementation of the *Apriori* algorithm (Agrawal and Srikant, 1994) in the *arules* package (Hahsler *et al.*, 2005; Hahsler *et al.*, 2018) to compute association rules with *NotFaulty* as target item (rule consequent). We set the threshold for the minimum support to 10% and the threshold for the minimum confidence to 90% (support and confidence are explained in Section 2.5.2.2). We experimented with different thresholds and these values produced good results (results for other configurations are in the dataset provided in Niedermayr *et al.* (2018a)). The minimum support avoids overly infrequent (*i.e.*, non-generalizable) rules from being created, and the minimum confidence prevents the creation of imprecise rules. Note that no rule (with *NotFaulty* as rule consequent) can reach a higher support than 50% after the SMOTE pre-processing. After computing the rules, we removed redundant ones using the corresponding function from the *apriori* package. We then sorted the remaining rules descending by their confidence.

Using these rules, we created two classifiers to identify low-fault-risk methods. They differ in the number of comprised rules. The strict classifier uses the top  $n$  rules until the share of faulty methods in all methods (of the training set) exceeds 2.5% in the LFR methods (of the training set). The more lenient classifier uses the top  $n$  rules until the share exceeds 5.0% in the LFR methods. (Example: We applied the top one rule to the training set, then applied the next rule, . . . , until the matched methods in the training set contained 2.5% out of all faults.) Figure 7.5 presents how an increase in the number of selected rules affects the proportion of LFR methods and the share of faulty methods that they contain. For RQ 1 and RQ 2, we computed the classifiers for each fold of each project. For RQ 3, we computed the classifiers once for each project.

To answer **RQ 1**, we used 10-fold cross-validation to evaluate the classifiers

---

<sup>1</sup> We computed the results for the empirical study once with and once without addressing the data imbalance in the training set. The prediction performance was better when applying SMOTE, therefore, we decided to use it.



Figure 7.5: Influence of the number of selected rules (LANG project). The number of rules influences the — proportion of low-fault-risk methods and the — share of faulty methods in LFR out of all faulty methods.

separately for each project. We computed the number and proportion of methods that were classified as “low fault risk” but contained a fault ( $\approx$  false positives). Furthermore, we computed the precision metric. Our main goal is to identify those methods that we can say, *with high certainty*, contain hardly any faults. Therefore, we consider it to be more important to achieve a high precision than to predict *all* methods that do not contain any faults in the dataset.

As the dataset is imbalanced with faulty methods in the minority, the proportion of faults in low-fault-risk methods might not be sufficient to assess the classifiers (SMOTE was applied only to the training set). Therefore, we further computed the *fault-density reduction*, which describes how much less likely the LFR methods contain a fault. For example, if 40% of all methods are classified as “low fault risk” and contain 10% of all faults, the factor is 4. It can also be read as: 40% of all methods contain only one fourth of the expected faults. We mathematically define the fault-density reduction factor based on methods as

$$\frac{\text{proportion of LFR methods out of all methods}}{\text{proportion of faulty LFR methods out of all faulty methods}}$$

and based on SLOC as

$$\frac{\text{proportion of SLOC in LFR methods out of all SLOC}}{\text{proportion of faulty LFR methods out of all faulty methods}}$$



For both classifiers (strict variant with 2.5%, lenient variant with 5.0%), we present the metrics for each project and the resulting median.

To answer **RQ 2**, we assessed how common methods classified as “low fault risk” are. For each project, we computed the absolute number of low-fault-risk methods, their proportion out of all methods, and their extent by considering their SLOC. *LFR SLOC* corresponds to the sum of SLOC of all low-fault-risk methods. The proportion of LFR SLOC is computed out of all SLOC of the project.

To answer **RQ 3**, we computed the association rules for each project using the methods of the other five projects as training data. We determined the number of used top  $n$  rules with the thresholds 2.5% and 5.0%. To allow a comparison with the within-project classifiers, we computed the same metrics as in RQ 1 and RQ 2.

To answer **RQ 4**, we computed for each method the 9 code and 15 change metrics that were used in [Giger et al. \(2012\)](#). The metrics and their descriptions are listed in Table 7.4. We applied *Random Forest* as machine-learning algorithm and configured it as in [Giger et al. \(2012\)](#). We computed the results for within-project predictions using 10-fold cross-validation and we further computed the results for cross-project predictions as in RQ 3. We present the same evaluation metrics as in the previous research questions.

### 7.3.5 Results

This section presents the results to the research questions. Data to reproduce the results is available at [Niedermayr et al. \(2018a\)](#).

**RQ 1: What is the precision of the classifier for low-fault-risk methods?** Table 7.5 presents the results. The methods classified to have low fault risk (LFR) by the stricter classifier, which allows a maximum fault share of 2.5% in the LFR methods in the (balanced) training data, contain between 2 and 8 faulty methods per project. The more lenient classifier, which allows

Table 7.4: RQ 4: Code and change metrics used in [Giger et al. \(2012\)](#).

Metric name	Description
<b>Code metrics</b>	
fanIN	Number of methods that reference a given method
fanOUT	Number of methods referenced by a given method
localVar	Number of local variables in the body of a method
parameters	Number of parameters in the declaration
commentToCodeRatio	Ratio of comments to source code (line-based)
countPath	Number of possible paths in the body of a method
complexity	McCabe cyclomatic complexity of a method
execStmt	Number of executable statements
maxNesting	Maximum nested depth of all control structures
<b>Change metrics</b>	
methodHistories	Number of times a method was changed
authors	Number of distinct authors that changed a method
stmtAdded	Sum of all statements added to a method
maxStmtAdded	Maximum number of statements added to a method
avgStmtAdded	Average number of statements added to a method
stmtDeleted	Sum of all statements deleted from a method
maxStmtDeleted	Maximum number of statements deleted from a method
avgStmtDeleted	Average number of statements deleted from a method
churn	Sum of churn (stmtAdded – stmtDeleted)
maxChurn	Maximum churn
avgChurn	Average churn
decl	Number of method declaration changes
cond	Number of condition expression changes in a method
elseAdded	Number of added else-parts in a method
elseDeleted	Number of deleted else-parts from a method

a maximum fault share of 5.0%, classified between 4 and 15 faulty methods as LFR. The median proportion of faulty methods in LFR methods is 0.3% and 0.4%, respectively.

The fault-density reduction factor for the stricter classifier ranges between 4.3 and 10.9 (median: 5.7) when considering methods and between 1.5 and 4.4 (median: 3.2) when considering SLOC. In the project LANG, 28.6% of all methods with 13.8% of the SLOC are classified as LFR and contain 4.1% of all faults; hence, the factor is 7.0 (SLOC-based: 3.4). The median factor for

Table 7.5: RQ 1, RQ2: Evaluation of within-project IDP to identify low-fault-risk methods.

Project	Faults in LFR		LFR methods		LFR methods		LFR SLOC		LFR methods contain ... % of all faults		Fault-density reduction (SLOC) (methods)	
	#	%	Prec.	Rec.	#	%	#	%	#	%	(methods)	(SLOC)
<b>Within-project IDP, 10-fold: min. support = 10%, min. confidence = 90%, rules until fault share in training set = 2.5%</b>												
Chart	4	0.1%	99.9%	44.1%	2,995	43.9%	11,228	15.8%	10.3%	10.3%	4.3	1.5
Closure	6	0.2%	99.8%	29.2%	3,759	28.9%	15,497	10.5%	4.1%	4.1%	7.1	2.6
Lang	3	0.5%	99.5%	29.6%	576	28.6%	2,242	13.8%	4.1%	4.1%	7.0	3.4
Math	2	1.1%	98.9%	18.4%	190	16.5%	570	4.8%	1.5%	1.5%	10.9	3.1
Mockito	5	0.6%	99.4%	35.1%	875	34.4%	6,128	25.1%	7.8%	7.8%	4.4	3.2
Time	8	0.1%	99.9%	80.4%	8,063	80.2%	62,063	78.1%	17.8%	17.8%	4.5	4.4
Median		0.3%	99.7%	32.3%		31.7%		14.8%	6.0%	6.0%	5.7	3.2
<b>Within-project IDP, 10-fold: min. support = 10%, min. confidence = 90%, rules until fault share in training set = 5.0%</b>												
Chart	4	0.1%	99.9%	44.8%	3,040	44.6%	11,563	16.3%	10.3%	10.3%	4.3	1.6
Closure	15	0.3%	99.7%	41.8%	5,385	41.5%	25,981	17.6%	10.1%	10.1%	4.1	1.7
Lang	6	0.7%	99.3%	45.0%	879	43.7%	3,630	22.3%	8.2%	8.2%	5.3	2.7
Math	7	2.7%	97.3%	24.3%	255	22.1%	878	7.3%	5.3%	5.3%	4.2	1.4
Mockito	6	0.5%	99.5%	47.8%	1,189	46.8%	8,260	33.8%	9.4%	9.4%	5.0	3.6
Time	9	0.1%	99.9%	82.8%	8,298	82.5%	63,333	79.7%	20.0%	20.0%	4.1	4.0
Median		0.4%	99.6%	44.9%		44.1%		20.0%	9.8%	9.8%	4.3	2.2

Table 7.6: Top five association rules for LANG (within-project, fold 1).

#	Rule	Support	Confidence ↓
1	{ <i>NoMethodInvocations</i> , <i>UniqueVariableIdentifiersLessThan2</i> } ⇒ { <i>NotFaulty</i> }	10.98%	100.00%
2	{ <i>NoMethodInvocations</i> , <i>SlocLessThan4</i> , <i>NoArithmeticOperations</i> } ⇒ { <i>NotFaulty</i> }	10.98%	100.00%
3	{ <i>NoMethodInvocations</i> , <i>SlocLessThan4</i> , <i>NoCastExpressions</i> } ⇒ { <i>NotFaulty</i> }	10.60%	100.00%
4	{ <i>NoMethodInvocations</i> , <i>NoArithmeticOperations</i> , <i>NoAssignments</i> , <i>NoNesting</i> } ⇒ { <i>NotFaulty</i> }	10.23%	100.00%
5	{ <i>NoMethodInvocations</i> , <i>NoArithmeticOperations</i> , <i>NoAssignments</i> , <i>NoIfConditions</i> } ⇒ { <i>NotFaulty</i> }	10.23%	96.43%

the more lenient classifier is 4.3 when considering methods and 2.2 when considering SLOC. The factor never falls below 1 for both classifiers.

*IDP can identify methods with low fault risk. On average, only 0.3% of the methods classified as “low fault risk” by the strict classifier are faulty. The identified LFR methods are, on average, 5.7 times less likely to contain a fault than an arbitrary method in the dataset.*

Table 7.6 exemplarily presents the top five rules for the project LANG. Low-fault-risk methods include, amongst others, methods that work with fewer than two variables and contain no method invocations as well as short methods without arithmetic operations, cast expressions, and method invocations.

**RQ 2: How large is the fraction of the code base consisting of methods classified as “low fault risk”?** Table 7.5 presents the results. The stricter classifier classified between 16.5% and 80.2% of the methods as LFR (median: 31.7%, mean: 38.8%), the more lenient classifier matched between 22.1% and 82.5% of the methods (median: 44.1%, mean: 46.9%). The median of the comprised SLOC in LFR methods is 14.8% (mean: 24.7%) respectively 20.0% (mean: 29.5%).

*Using within-project IDP, on average, 32–44% of the methods, comprising approximately 15–20% of the SLOC, can be assigned a lower importance during testing. In the best case, when ignoring 16.5% of the methods, it is still possible to catch 98.5% of the faults (MATH).*

**RQ 3: Is a trained classifier for methods with low fault risk generalizable to other projects?** Table 7.7 presents the results for cross-project prediction with training data from the respective other projects. Compared to the results of the within-project prediction, except for MATH, the number of faults in LFR methods decreased or stayed the same in all projects for both classifier variants. While the median proportion of faults in LFR methods slightly decreased, the proportion of LFR methods also decreased in all projects except MATH. The median proportion of LFR methods is 23.3% (SLOC: 8.1%) for the stricter classifier and 26.3% (SLOC: 12.6%) for the more lenient classifier.

The fault-density reduction improved compared to the within-project prediction for the method and SLOC level in both classifier variants: For the stricter classifier, the median of the method-based factor is 10.9 (+5.2), and the median of the SLOC-based factor is 3.9 (+0.7). Figures 7.6 illustrates the fault-density reduction for both within-project (RQ 1, RQ 2) and cross-project (RQ 3) prediction.

If the rules were computed on data of *all six* projects, the strict classifier would work with 10 rules and the lenient one with 46 rules. Table 7.8 presents the top five rules thereof. In this set, rule 2 constitutes a more

Table 7.7: RQ3: Evaluation of cross-project IDP.

Project	Faults in LFR	LFR methods	LFR methods	LFR SLOC	LFR methods contain...% of all faults	Fault-density reduction (methods)	(SLOC)				
	#	%	Prec.	Rec.	#	%	#				
<b>Gross-project IDP: min. support = 10%, min. confidence = 90%, rules until fault share in training set = 2.5%</b>											
Chart	3	0.1%	99.9%	32.1%	2,182	32.0%	7,434	10.5%	7.7%	4.2	1.4
Closure	2	0.1%	99.9%	25.0%	3,207	24.7%	11,584	7.9%	1.4%	18.3	5.8
Lang	1	0.2%	99.8%	23.1%	449	22.3%	1,357	8.3%	1.4%	16.3	6.1
Math	8	2.9%	97.1%	26.6%	280	24.3%	1,129	9.4%	6.1%	4.0	1.6
Mockito	1	0.2%	99.8%	21.7%	539	21.2%	1,698	6.9%	1.6%	13.6	4.4
Time	1	0.1%	99.9%	18.4%	1,845	18.3%	5,807	7.3%	2.2%	8.3	3.3
Median		0.2%	99.8%	24.0%		23.3%		8.1%	1.9%	10.9	3.9
<b>Gross-project IDP: min. support = 10%, min. confidence = 90%, rules until fault share in training set = 5.0%</b>											
Chart	4	0.2%	99.8%	35.5%	2,411	35.4%	9,363	13.2%	10.3%	3.4	1.3
Closure	4	0.1%	99.9%	25.9%	3,327	25.6%	15,583	10.6%	2.7%	9.5	3.9
Lang	4	0.7%	99.3%	27.7%	542	26.9%	1,959	12.0%	5.5%	4.9	2.2
Math	18	5.1%	94.9%	32.9%	354	30.7%	1,634	13.7%	13.6%	2.2	1.0
Mockito	1	0.2%	99.8%	25.0%	620	24.4%	3,495	14.3%	1.6%	15.6	9.1
Time	1	0.0%	100.0%	20.0%	2,007	20.0%	7,552	9.5%	2.2%	9.0	4.3
Median		0.2%	99.8%	26.8%		26.3%		12.6%	4.1%	6.9	3.1

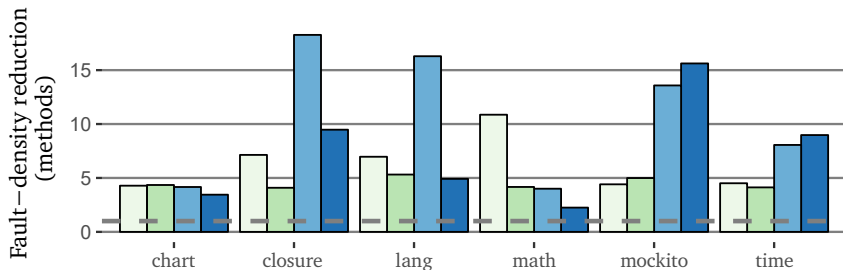


Figure 7.6: Comparison of IDP within-project ( 2.5%, 5.0%) with IDP cross-project ( 2.5%, 5.0%) classifiers (method-based). The fault-density reduction expresses how much less likely a LFR method contains a fault (definition in 7.3.4). Higher values are better. (Example: If 40% of the methods are LFR and contain 5% of all faults, the factor is 8.) The dashed line is at one; no value falls below.

Table 7.8: Top five association rules computed on all projects.

#	Rule	Support	Confidence ↓
1	{ <i>NoMethodInvocations</i> , <i>SlocLessThan4</i> , <i>NoArithmeticOperations</i> , <i>NoNullLiterals</i> } ⇒ { <i>NotFaulty</i> }	10.43%	96.76%
2	{ <i>NoMethodInvocations</i> , <i>SlocLessThan4</i> , <i>NoArithmeticOperations</i> } ⇒ { <i>NotFaulty</i> }	11.03%	96.09%
3	{ <i>NoMethodInvocations</i> , <i>SlocLessThan4</i> , <i>NoCastExpressions</i> , <i>NoNullLiterals</i> } ⇒ { <i>NotFaulty</i> }	10.43%	95.43%
4	{ <i>NoMethodInvocations</i> , <i>SlocLessThan4</i> , <i>NoCastExpressions</i> , <i>NoInstantiations</i> } ⇒ { <i>NotFaulty</i> }	10.13%	95.31%
5	{ <i>NoMethodInvocations</i> , <i>SlocLessThan4</i> , <i>NoCastExpressions</i> } ⇒ { <i>NotFaulty</i> }	11.03%	94.85%

general form of rule 1, thereby being more frequent and less precise. The same applies to rules 4 and 5. The rules are similar to the ones presented in Table 7.6, confirming their generalizability.

*Using cross-project IDP, on average, 23–26% of the methods, comprising approximately 8–13% of the SLOC, can be classified as “low fault risk.” The methods classified by the stricter classifier contain, on average, fewer than one 11<sup>th</sup> of the expected faults.*

**RQ 4: How does the classifier perform compared to a traditional defect prediction approach?** Table 7.9 presents the results of the within- and cross-project prediction according to the approach by Giger *et al.* (2012). In the within-project prediction scenario, the classifier predicts on average 99.4% of the methods to be non-faulty. Consequently, the average recall for non-faulty methods reaches 99.9%. However, the number of methods that are classified as non-faulty but actually contain a fault increases by orders of magnitude compared to the IDP approach (*i.e.*, precision deteriorates). For example, 77% of CLOSURE’s faulty methods are wrongly classified as non-faulty. The median fault-density reduction is 1.6 at the method level (strict IDP: 5.7) and 1.4 when considering SLOC (strict IDP: 3.2). Consequently, methods classified by the traditional approach as having a low fault risk are still less likely to contain a fault than other methods, but the performance falls behind the one achieved by the IDP classifiers.

The results in the cross-project prediction scenario are similar. In four of the six projects, the number of faults in LFR methods increased compared with the within-project prediction scenario. The fault-density reduction deteriorated to 1.2 both at the method and SLOC level (for comparison, strict IDP achieves 10.9 and 3.9, respectively). In all projects, IDP outperformed the traditional approach.



Table 7.9: RQ 4: Results of a traditional defect prediction approach.

Project	Faults in LFR		LFR methods		LFR methods		LFR SLOC		LFR methods		Fault-density reduction	
	#	%	Prec.	Rec.	#	%	#	%	#	%	(methods)	(SLOC)
<b>Within-project defect prediction: traditional approach used in Giger et al. (2012)</b>												
Chart	36	0.5%	99.5%	99.9%	6,785	99.9%	70,457	99.6%	92.3%	1.1	1.1	1.1
Closure	114	0.9%	99.1%	99.9%	12,862	99.6%	142,518	97.6%	77.0%	1.3	1.3	1.3
Lang	36	1.9%	98.1%	99.5%	1,905	97.6%	14,462	91.2%	49.3%	2.0	1.9	1.9
Math	62	6.4%	93.6%	98.4%	967	91.9%	7,234	66.5%	47.0%	2.0	1.4	1.4
Mockito	46	1.9%	98.1%	99.8%	2,481	99.1%	24,232	99.5%	71.9%	1.4	1.4	1.4
Time	26	0.3%	99.7%	100.0%	9,995	99.8%	78,809	99.8%	57.8%	1.7	1.7	1.7
Median		1.4%	98.6%	99.9%		99.4%		98.6%	64.8%	1.6	1.4	1.4
<b>Cross-project defect prediction: traditional approach used in Giger et al. (2012)</b>												
Chart	32	0.5%	99.5%	99.2%	6,755	99.1%	68,214	96.3%	82.1%	1.2	1.2	1.2
Closure	82	0.6%	99.4%	99.5%	12,859	99.0%	141,322	96.0%	55.4%	1.8	1.7	1.7
Lang	52	2.6%	97.4%	99.9%	1,990	98.9%	15,085	92.8%	71.2%	1.4	1.3	1.3
Math	103	9.2%	90.8%	99.8%	1,123	97.3%	9,512	79.5%	78.0%	1.2	1.0	1.0
Mockito	58	2.3%	97.7%	100.0%	2,534	99.8%	24,420	99.8%	90.6%	1.1	1.1	1.1
Time	39	0.4%	99.6%	100.0%	10,050	99.9%	79,191	99.7%	86.7%	1.2	1.2	1.2
Median		1.5%	98.5%	99.9%		99.0%		96.1%	80.0%	1.2	1.2	1.2

### 7.3.6 Threats to Validity

Next, we discuss the threats to internal and external validity.

*Threats to internal validity:* The learning and evaluation was performed on information extracted from Defects4J (Just *et al.*, 2014b). Therefore, the quality of our data depends on the quality of Defects4J. The following problems are common for defect datasets created by analyzing changes in commits that reference a bug ticket in an issue-tracking system. First, commits that fix a fault but do not reference a ticket in the commit message cannot be detected (Bachmann *et al.*, 2010). Consequently, the set of commits that reference a bugfix may not be a fair representation of all faults (Bird *et al.*, 2009; D'Ambros *et al.*, 2012; Giger *et al.*, 2012). Second, bug tickets in the issue tracker may not always represent faults and vice versa. Antoniol *et al.* (2008) and Herzig *et al.* (2013) pointed out that a significant number of tickets in the issue trackers of open-source projects is misclassified. Therefore, it is possible that not all bugfix commits were spotted. Third, methods may contain faults that have not been detected or fixed yet. It is generally not possible to prove that a method does not contain any faults. Fourth, a commit may contain changes (such as refactorings) that are not related to the bugfix, but this problem does not affect the Defects4J dataset due to the authors' manual inspection. These threats are present in nearly all defect prediction studies, especially in those operating at the method level. Defect prediction models were found to be resistant to such kind of noise to a certain extent (Kim *et al.*, 2011).

*Threats to external validity:* The empirical study was performed with six mature open-source projects written in Java. The projects are libraries and their results may not be applicable to other application types, for example, large industrial systems with user interfaces. The results may also not be transferable to projects of other languages, for the following reasons: First, Java is a strongly typed language that provides type safety. It is unclear whether the IDP approach works for languages without type safety because it

could be that even simple methods in such languages exhibit a considerable number of faults. Second, in case the approach as such is applicable to other languages, the collected metrics and the low-fault-risk classifier need to be validated and adjusted. Other languages may use language constructs in a different way or use constructs that do not exist in Java. For example, a classifier for the C language should take constructs such as GOTOs and the use of pointer arithmetic into consideration. Furthermore, the projects in the dataset, which was published in 2014, did not contain code with lambda expressions introduced in Java 8 (Neward, 2013). Therefore, in newer projects that make use of lambda expressions, the presence of lambdas should be taken into consideration when classifying methods. Consequently, further studies are necessary to determine whether the results are generalizable.

As in many defect prediction studies (Madeyski and Jureczko, 2015; Ostrand *et al.*, 2004), we did not take the severity of a fault into account and made the assumption that each fault has the same relevance. Reasons for this assumption are that the severity of a bug ticket is often highly subjective (Ostrand *et al.*, 2004) and that it is not possible with reasonable effort to objectively determine failure follow-up costs caused by a fault. The latter in particular applies to open-source libraries, which can be integrated as dependencies into other systems; a fault can trigger further failures in the surrounding system. In practice, not all faults have the same importance because some cause higher failure follow-up costs than others.

## 7.4 Discussion

The results of our empirical study show that only very few low-fault-risk methods actually contain a fault. Hence, we can conclude that IDP can successfully identify methods that are not fault-prone. On average, 31.7% of the methods (14.8% of the SLOC) matched by the strict classifier contain only 6.0% of all faults, resulting in a considerable fault-density reduction for the matched methods. In any case, low-fault-risk methods are less fault-prone than other methods (fault-density reduction is higher than one in all

projects); based on methods, LFR methods are at least twice less likely to contain a fault. For the stricter classifier, the extent of the matched methods, which could be deferred in testing, is between 5% and 78% of the SLOC of the respective project. The more lenient classifier matches more methods and SLOC at the cost of a higher fault proportion, but still achieves satisfactory fault-density reduction values. This shows that the balance between fault risk and matched extent can be influenced by the number of considered rules to reflect the priorities of a software project.

Interestingly, the cross-project IDP classifiers, which are trained on data from the respective other five projects, exhibit higher precision than the within-project IDP classifiers. Except for the MATH project, the LFR methods contain fewer faulty methods in the cross-project prediction scenario. This is in line with the method-based fault-density reduction factor of the strict classifier, which in four of six cases is better in the cross-project scenario (SLOC-based: three of six cases). However, the proportion of matched methods decreased compared to the within-project prediction in most projects. Accordingly, the cross-project results suggest that a larger, more diversified training set identifies LFR methods more conservatively, resulting in a higher precision and a lower extent of matched methods.

MATH is the only project in which IDP within-project prediction outperformed IDP cross-project prediction. This project contains many methods with mathematical computations expressed by arithmetic operations, which are often wrapped in loops or conditions; most of the faults are located in these methods. Therefore, the within-project classifiers used a few very precise rules for identifying LFR methods.

In sum, the results show that the IDP approach can be used to identify methods that are, due to the “triviality” of their code, less likely to contain any faults. Hence, these methods require less focus during QA activities. Depending on the criticality of the system and the risk one is willing to take, the development of tests for these methods can be deferred or even omitted in case of insufficient available test resources. Furthermore, these methods do not pose a threat when they are pseudo-tested.

The results also suggest that IDP is applicable in cross-project prediction

scenarios, indicating that characteristics of low-fault-risk methods differ less among projects than characteristics of faulty methods do. Therefore, IDP is applicable in (new) projects with no (precise) historical fault data to prioritize the code to be tested.

#### 7.4.1 Limitations

While low-fault-risk methods are less fault-prone, it is not guaranteed that they never contain any fault. An inspection of faulty methods incorrectly classified as having low fault risk showed that some faults were fixed by only adding further statements (e.g., to handle special cases). This means that even if the (existing) code of a method is not faulty, a method can still be faulty due to missing code. Further imaginable examples of faulty low-fault-risk methods are simple getters that return the wrong variable or completely empty methods that have been unintentionally left empty. Therefore, while these methods are much less fault-prone, it cannot be assumed that they never contain any fault. Consequently, excluding low-fault-risk methods from testing and other QA activities carries a risk.

#### 7.4.2 Applications

The IDP classifier can identify low-fault-risk methods, which—depending on the criticality of an application—may be considered as *too trivial to test*; that is, they may not warrant being tested. Such methods constitute only a very low risk when they are pseudo-tested and therefore, it is not rewarding to determine how effectively a low-fault-risk method is tested when it is covered by tests. Consequently, the IDP classifier can be used to exclude trivial methods from a mutation analysis, which makes it possible to focus on the more relevant methods and helps reduce analysis scope and duration.

Another scenario for an automated exclusion of trivial methods could be the *Test Gap Analysis (TGA)* (Juergens and Pagano, 2016; Rott *et al.*, 2017). This analysis shows testers and test managers which methods were added or changed during the implementation of a new feature or a bugfix,

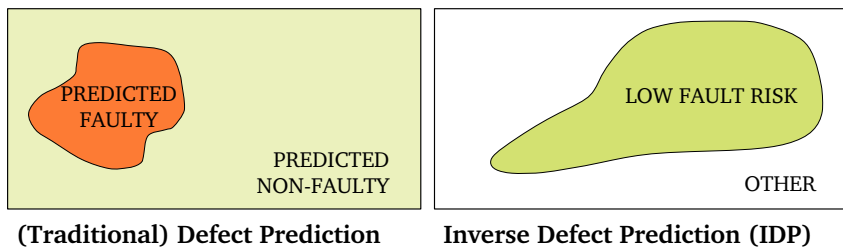


Figure 7.7: Defect prediction and IDP have different classification classes. Ideally, low-fault-risk methods are not contained in the set of faulty methods. As a second priority, they should form a possibly large subset of non-faulty methods to increase the savings potential in QA activities.

but not covered by any automated or manual test after the modification. Such recently modified but untested methods are much more fault-prone, according to [Eder et al. \(2013\)](#). However, not all methods that constitute test gaps have the same complexity and fault-proneness. Therefore, filtering out trivial ones makes it possible to focus on the more relevant test gaps.

### 7.4.3 Relation to Defect Prediction

IDP presents another view on defect prediction. Like traditional defect prediction, IDP also uses a set of metrics to characterize artifacts, applies transformations to pre-process metrics, and uses a machine-learning classifier to build a prediction model. As can be seen in [Figure 7.7](#), the difference rather lies in the predicted classes and the underlying target. While defect prediction classifies an artifact either as *buggy* or *non-buggy*, IDP identifies only those methods that exhibit a *low fault risk* with high certainty and does not make an assumption about the remaining methods, for which the fault risk is at least medium or cannot be reliably determined. Consequently, the objective of the prediction also differs. Defect prediction aims to achieve a high recall to detect as many faults as possible, and a high precision to minimize the number of false positives. IDP strives to achieve high precision

to ensure that low-fault-risk methods contain indeed hardly any faults (*i.e.*, IDP should classify hardly any faulty methods erroneously as “low fault risk”). To enable focusing on precision, IDP does not necessarily seek to identify all non-faulty methods; that is, recall is only second priority. When predicting faulty methods in traditional defect prediction, low recall of faulty methods would be a major issue because it could lead to omitted methods that are erroneously not classified as faulty. In IDP, lower recall of low-fault-risk methods reduces possible savings, but does not lead to faulty methods being omitted. Still, it is desirable for IDP to achieve a sufficiently high recall so that reasonable savings potential arises when treating low-fault-risk methods with a lower priority in QA activities.

## 7.5 Summary

Developer teams and testers often face the problem of scarce test resources, and therefore need to prioritize their testing efforts (*e.g.*, when writing new automated unit tests or when addressing identified test gaps). Defect prediction can support developers in this activity. We propose an inverse view on defect prediction to identify methods that are so “trivial” that they contain hardly any faults. We studied how unerringly such low-fault-risk methods can be identified, how common they are, and whether the proposed approach is applicable for cross-project predictions.

We show that IDP using association rule mining on code metrics can successfully identify low-fault-risk methods. The identified methods contain considerably fewer faults than the average code and can provide a savings potential for QA activities. Depending on the parameters, a lower priority for QA can be assigned on average to 31.7% (strict classifier) respectively 44.1% (lenient classifier) of the methods, amounting to 14.8% respectively 20.0% of the SLOC. While cross-project defect prediction is a challenging task (He *et al.*, 2012; Zimmermann *et al.*, 2009), our results suggest that the IDP approach can be applied in a cross-project prediction scenario at the method level. In other words, an IDP classifier trained on one or more (Java

open-source) projects can successfully identify low-fault-risk methods in other Java projects for which no fault data (or no precise fault data) exists.





# ENHANCED TEST CASE SELECTION AND PRIORITIZATION

*This chapter devises a hybrid test case selection and prioritization approach that is enhanced with information about test effectiveness.*

## 8.1 Motivation

Automated software tests help identify faults early and should ideally be executed after each integration of code changes into the main repository. With the introduction of agile development methodologies, the number of automated test cases rises in many software projects and a growing number of projects exhibits a large test suite (Madeyski, 2009).

The increase in the number of test cases in a test suite is linked to a longer execution duration. In large industrial systems, the execution of the whole test suite often takes days or even weeks (Rothermel *et al.*, 1999; Zhang, 2018), particularly if tests involve interactions with hardware components or complex user interfaces. At one large German car manufacturer, for example, the execution of all hardware-in-the-loop (HIL) tests of a certain automotive component takes 20 days.

Consequently, continuous integration is impeded. Even with the use of parallelization, it is not feasible to execute the whole test suite after each committed code change. This leads to situations in which large test suites are executed only nightly, at weekends, or even less often, so that the time span between the inadvertent introduction of a fault and its detection rises. This leads to negative consequences. First, developers usually start new development activities on the latest code version, which will probably not have been examined by tests. It may be a “broken” version containing faults that would have been revealed by a test suite execution. Duvall *et al.* (2007) stated that “problems causing failing tests need to be fixed before they propagate and cause other failures.” Second, efforts to fix a test failure increase at a later detection. This is because it is more difficult at a later point to narrow down the changes that caused a test failure. Furthermore, developers who debug a test failure that is caused by changes they conducted some time ago need to familiarize themselves with their changes again (Muşlu *et al.*, 2015).

One possible solution to enable continuous integration for long-running test suites is the use of test case selection techniques. Test case selection identifies which test cases of a test suite are relevant for testing a given set of

code changes (Yoo and Harman, 2012). Moreover, quick feedback from the test execution in case of failing tests is desirable. Duvall *et al.* (2007) stated that “one wants fast builds and builds to fail fast.” Similarly, Vocke (2018) wrote that “a good build pipeline tells you that you messed up as quick as possible.” Test case prioritization techniques facilitate that: they order test cases in such a way that a certain goal is achieved as quickly as possible; for example, identifying the first failure early (Rothermel *et al.*, 2001).

Hybrid approaches combine selection and prioritization, thus reducing the number of test cases to be executed and ordering the selected tests (Wong *et al.*, 1997). Most existing test case selection and prioritization approaches rely on code coverage. Given that several studies have shown that code coverage is a questionable predictor of test effectiveness, approaches based solely on coverage may not work in an ideal way. Therefore, an interesting question is whether test case selection and prioritization can be improved by further integrating information about test effectiveness. This chapter looks at whether such an approach, which is aware of pseudo-tested test-case method pairs, achieves a shorter time to the detection of the first test failure.

*The main contributions of this chapter are:* We devise, implement, and evaluate a hybrid test case selection and prioritization approach that additionally incorporates test effectiveness information to reduce the time to the first test failure in a test suite execution.

## 8.2 Existing Approach

*Teamscale* is a software intelligence platform (Haas *et al.*, 2019c) that incrementally analyzes data from version control systems and combines it with further data that accrue during the software development process (e.g., code coverage from test executions, issues from the issue tracker, and more). *Teamscale* aims to create transparency on a system’s code quality, architecture, and development process towards developers, testers, and managers.

Besides various static code analyses, *Teamscale* provides a feature called

*Test Impact Analysis (TIA)*, which selects and prioritizes test cases in respect to given code changes, with the goal of minimizing the time to the first failure in a test suite execution. We implemented the prioritization logic inspired by a prototype from Dreier (2017).

Below, we describe how the existing test case selection and prioritization logic works. An overview of the computation steps is presented in Figure 8.1.

### 8.2.1 Required Input Data

The existing approach requires the following input data.

*Source code and changes:* Teamscale is connected to the repositories of a software project and periodically fetches new commits representing code changes. A shallow parser parses the source code, making it possible to work on structural entities (e.g., methods or statements). Changes can be mapped to these entities.

*Test-case specific coverage:* Within his master's thesis, Dreier (2017) developed the *teamscale-jacoco-agent*,<sup>1</sup> which extends the coverage recorder *JaCoCo* and allows coverage to be collected separately per test case. The agent can be employed as Maven or Gradle plug-in and thereby integrated into the build process. It listens to test-start and test-end events and separates recorded coverage accordingly. The agent generates coverage reports in the form of an XML file as output. Besides coverage information, it also contains the duration of each test case as measured by the test runner. The report can be uploaded to Teamscale, which processes the data and stores for each method its covering test cases.

### 8.2.2 Test Case Selection

The test case selection step identifies all test cases that cover code that has been changed in a certain time span. The time span may include, for example, changes conducted since the last release or the last test execution

---

<sup>1</sup> <https://github.com/cqse/teamscale-jacoco-agent>

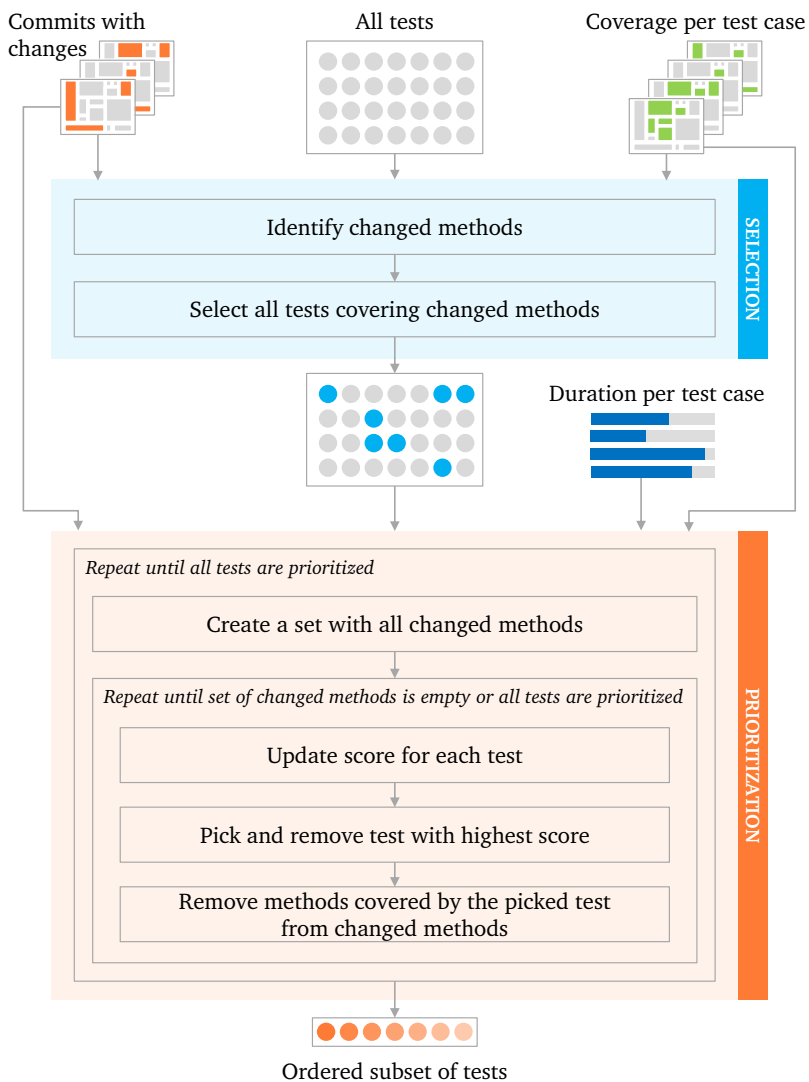


Figure 8.1: Overview of the steps of the Test Impact Analysis (TIA).

until now. All commits conducted in the time span are selected, and methods that were added or changed in these commits are collected.<sup>1</sup> Based on the uploaded test-case specific code coverage, all test cases covering at least one of the changed methods are selected.

Changed test cases that are represented as methods in the code will also appear in the set of added or changed methods. Consequently, changed test cases, for which code coverage information may be outdated due to changes to the test, are also selected.

### 8.2.3 Test Case Prioritization

The prioritization step takes the selected test cases as input<sup>2</sup> and orders them so that changed methods are covered as quickly as possible. The prioritization algorithm is round-based and similar to the concept used at Microsoft, which was presented by [Srivastava and Thiagarajan \(2002\)](#).

The idea behind this concept is to iteratively choose the test case that provides the best ratio of additional coverage per test duration to coverage achieved by already picked tests. This ratio is assigned as a score to each as yet un-prioritized test case. The test case with the highest score is chosen and the scores of all remaining test cases are updated. To facilitate the computation of the additional coverage of a test case, changed methods that are covered by already picked tests are marked. Once all changed methods that can be covered are marked, a new prioritization round begins, in which these markers are reset. The second prioritization round is special; in this round, all added and changed test cases are scheduled.

This approach aims to cover changed methods as quickly as possible and is an application of the set cover problem ([Dreier, 2017](#)). The implementation uses a greedy algorithm to solve this problem known to be  $\mathcal{NP}$ -complete.

---

<sup>1</sup> In the following, when we refer to changed methods, we mean both added and changed methods.

<sup>2</sup> It is also possible to skip the selection and apply only the prioritization step. In this case, all available test cases are considered.

### 8.2.4 Provision of Results

Teamscale provides a web service that allows the selection and prioritization results to be retrieved in the form of an ordered list of test cases. Test runners can fetch this list and execute test cases accordingly.

## 8.3 Enhanced Approach with Test Effectiveness Information

Our enhanced approach extends the existing TIA approach, with the goal of further reducing the time to the first test failure. For that, the approach additionally considers a test's fault detection capabilities.

The existing and enhanced approaches both order test cases by a score. The existing approach determines the score of a test by computing the number of additionally covered methods (to methods covered by other already prioritized test cases) in respect to the test's duration. Each additionally covered method contributes equally to the score.

By contrast, the enhanced approach assigns a weight to each test-case method relation, which reflects whether the given method is (predicted to be) pseudo-tested by the given test case. Therefore, the enhanced approach needs pseudo-testedness information at the level of test-case method relations. It can work with computed pseudo-testedness information stemming from a mutation analysis, or alternatively use predicted information from a machine-learning model based on the minimal stack distance. The enhanced approach needs to take into account that provided information about pseudo-testedness might contain inaccuracies. This is because mutation testing verdicts will have been computed on an earlier commit, results may contain equivalent mutants, metric values used in prediction models may be outdated, and prediction models generally do not achieve perfect accuracy. The approach accounts for that by assigning pseudo-tested test-case method pairs a lower weight instead of excluding them completely. Consequently, an incomplete or partially outdated mutation matrix may reduce its positive influence on selection and prioritization, but does not necessarily cause harm.

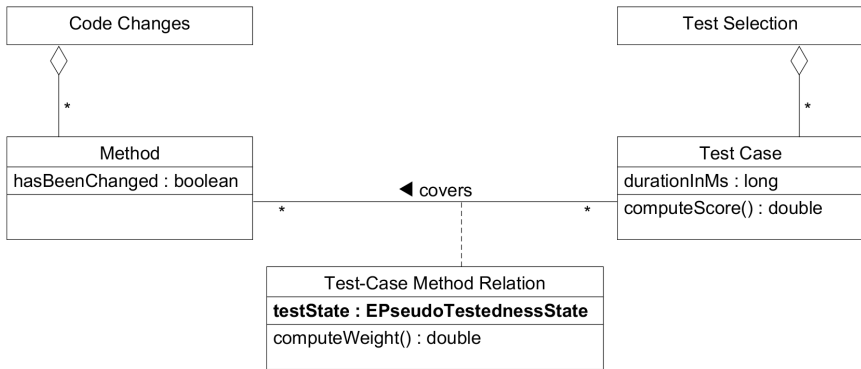


Figure 8.2: Class diagram illustrating how pseudo-testedness information influences the score that a test case achieves by covering a method.

The weights for test-case method pairs were determined in a preliminary study. They are for pseudo-testedness information from a mutation analysis:

- Method known to be not pseudo-tested by a given test case: 1.0
- Method known to be not pseudo-tested, but killing test cases are unknown (this occurs when a mutant is killed due to a timeout or memory error): 0.15
- Method known to be pseudo-tested by a given test case: 0.05
- Method with no available pseudo-testedness information (this occurs when a method was not mutated due to exclusions or when a method did not exist in the analyzed revision): 0.15

For pseudo-testedness information from a prediction model, the weights account for higher inaccuracies. The used weights are:

- Method predicted to be not pseudo-tested by a given test case: 1.0
- Method predicted to be pseudo-tested by a given test case: 0.2
- Method with no available information (this occurs when a method's measures needed for a prediction are not available): 0.7



Figure 8.2 summarizes the relations between the relevant entities and visualizes where pseudo-testedness information is used.

## 8.4 Evaluation

We evaluated the performance of the enhanced test case selection and prioritization approach and compared it with the performance of the existing approach and of a regular test execution. The goal was to find out whether the enhanced approach can reduce the time until the first test failure in a test suite execution when a project contains a substantial proportion of pseudo-tested methods.

*Procedure:* The evaluation was done on the project `PDFBOX`. We selected this study object because it is a well-known multi-module project that contains 1,587 test cases achieving a line coverage of 49.7%, but 26.3% of the covered methods are pseudo-tested (see Table 6.4 in Section 6.4.5). The project is open-source and hosted on GitHub.<sup>1</sup>

Starting with revision `d9930344`, we computed a full mutation matrix with the Descartes operator, minimal stack distance values, further measures needed to predict pseudo-tested methods, and test-case specific code coverage. The mutation matrix contained 2,867 mutated methods and 622 test cases. A single test case covered 81.4 methods, on average, and detected mutations in approximately 67% of its covered methods.

Over 90 following commits containing at least one code change were iterated. In each commit, we randomly marked one of the changed methods as faulty and derived the thereby failing test cases from the mutation matrix. *Teamscale* was used to compute the test case selection and prioritization for the code changes in the analyzed commit using the existing approach, the enhanced approach with computed mutation data, and the enhanced approach with a prediction model for pseudo-testedness. In `PDFBOX`, the prediction model at the level of test-case method pairs achieved 83.8% precision

---

<sup>1</sup> <https://github.com/apache/pdfbox>

Table 8.1: Comparison of different approaches in respect to the number of test cases and their duration until the first test failure.

Approach	Test index		Duration (ms)	
	mean	median	mean	median
Regular Maven execution	284.35	218	72,263	62,300
Existing TIA	6.28	1	1,775	51
Enhanced TIA (mutation data)	2.95	1	1,118	15
Enhanced TIA (predictions)	5.98	1	1,511	23

and 83.8% recall (see Table 6.8 in Section 6.4.5). We also determined the test case ordering used in a regular Maven execution. Next, we identified in the ordered test case list the first test case that is known to fail for the method marked as faulty. We computed the index of that test case and the sum of the duration of the test cases until and including that first failing one. We refer to the duration as the “time to first failure (TFF).”

*Results:* Table 8.1 presents the mean duration and TFF of each approach. According to the results, the existing TIA approach without test effectiveness information already put one of the failing test cases first in 55 of 90 analyzed commits. The enhanced approach using test effectiveness information from a mutation analysis outperformed the existing approach in 25 of the remaining 35 commits, in which it achieved a lower TFF and a prior position for the first failing test case. For example, in commit 18f22894, the first failing test case was on index 70 using the existing TIA approach, while it was on index 13 using the enhanced approach with mutation data. The enhanced approach using test effectiveness prediction also achieved better results than the existing approach (in 8 commits). In the analyzed commits, neither enhanced approach ever performed worse than the existing approach.

As expected, all selection and prioritization approaches clearly outperformed a regular test suite execution. The regular test suite execution (as conducted by Maven) took 284 test cases on average, with a duration of 72,263 ms until the first test failure, which corresponds to 32.9% of the

execution time of the whole test suite. On average, test cases selected and ordered by the existing TIA approach reached the first test failure after 0.81% of the duration of all test cases. Test case orderings by the enhanced approach exhibited a mean duration of 0.51% when using mutation data and 0.69% when using prediction data. Consequently, the enhanced approach could reduce the TFF by further 37% compared to the existing approach.

*Threats to validity:* We determined the time to the first failure of the approaches by summing up the duration of the test cases. However, test cases may not be executable in the provided order. This may be due to tacit dependencies between test cases, given that a set of test cases might need to be executed in a certain order to succeed. It may also be due to limitations in the test framework, which may not allow a single test case in a class to be executed, or test cases from different modules to be executed arbitrarily.

In this evaluation, we examined only one study object and used a mutation matrix both as input for the approach and as an evaluation means. Therefore, the external validity is limited. The outcome may apply to other Java projects exhibiting similar characteristics (in particular, a substantial proportion of pseudo-tested methods), but further studies are necessary to determine the generalizability of the results.

## 8.5 Limitations

This approach identifies changes conducted to code artifacts, and proposes test cases based on code coverage enriched with test effectiveness information. However, test cases may also fail for reasons other than code changes. Non-code files such as configuration or test data files are not tracked and changes to these files may cause failures in tests that are not selected by the approach because no coverage relation exists between the tests and non-code files. Moreover, test cases may also fail because of the test environment; for example, when a web-service accessed by an integration test is not available due to network problems. Similarly, non-determinism in test cases results in flickering tests, which sporadically fail independent of code changes. Non-

deterministic test executions also lead to unstable code coverage information, hampering test case selection and prioritization. These limitations apply to most coverage-based test case selection and prioritization approaches; they are not specific to our enhanced approach.

Test effectiveness information may change over time, especially when test cases are improved or major changes are conducted to the covered code. Therefore, pseudo-testedness information or collected measures to build a prediction model need to be updated occasionally. The more often this information is updated, the more accurate it will be, resulting in a better performance of the approach. However, collecting or incrementally updating this information involves considerable costs, which conflict with the goal of reducing efforts for test suite executions.

## 8.6 Summary

We extended an existing test case selection and prioritization approach, which considers change information and test-case specific code coverage. This enhanced approach also incorporates pseudo-testedness information and assigns a weight to test-case method relations, which reflects whether a test case can fail because of faults in that method.

The evaluation shows that the enhanced approach, using either computed mutation data or a prediction model, can further reduce the time to the first test failure in a test suite execution.

A limitation that applies to both the existing and enhanced approach is that they propose test cases based on code changes, but test failures may also be caused by changes to non-code artifacts, problems in the test environment, and non-determinism.

CHAPTER  
9

# CONCLUSION AND FUTURE WORK

*This chapter summarizes and discusses this dissertation and its limitations, and explores what future work could be undertaken on the basis of the completed work.*

## 9.1 Discussion

We have presented a mutation approach to identify pseudo-tested methods, which are covered by test cases but untested in their entirety. One advantage of this approach is that it is substantially faster than mutation testing with traditional mutation operators because it strongly reduces the number of generated mutants. Furthermore, this approach does not suffer from the problem of equivalent mutants due to the design of the mutation operator. The outcome of the analysis—a list of methods lacking effective testing—is easy for developers to interpret and action.

The uncovered pseudo-tested methods exist in all analyzed study objects and an analysis of them has shown that their lack of test effectiveness is relevant. This was confirmed by [Vera-Pérez et al. \(2017\)](#).

Although no single measure strongly correlates with the mutation testing verdict of a method, we have presented that a combination of different measures in a random forest model can successfully identify pseudo-tested methods as well as pseudo-tested test-case method pairs. The computation of the required measures for the model is cheaper than a mutation analysis, by orders of magnitude. We suggest that such prediction models can be applied as a preceding, less costly step to mutation testing or be used in scenarios where mutation testing is not applicable due to technical limitations.

Our analysis of the fault risk of methods confirms that not all methods exhibit the same fault risk. Consequently, we feel it is sensible to treat methods differently depending on how fault-prone they are. In particular, we propose excluding low-fault-risk methods from a mutation analysis to reduce analysis efforts and increase the relevance of the gained results. In addition, low-fault-risk methods can be filtered out from test gaps, which are added or changed methods that have not been tested after the last modification, to reduce efforts for implementing automated or conducting manual tests.

Finally, we have presented an approach that integrates test effectiveness information into an existing hybrid test case selection and prioritization approach. Such an enhanced approach can lead to a better prioritized test

list in respect to given changes, resulting in a lower time to the first failure in test suite executions.

## 9.2 Limitations

One limitation of the mutation operator to identify pseudo-tested methods is that it is not as precise as other mutation operators. It operates at the method level and can detect when a method in its entirety is not tested. Traditional mutation operators operate at the statement level and generate subtler faults, which are more difficult to detect than the invasive transformation of our mutation operator. Consequently, the fact that a method is not pseudo-tested does not imply that all of its statements are effectively tested. This is by design and acknowledged to achieve benefits, such as, the reduced run-time complexity or the avoidance of equivalent mutants.

Another limitation concerns the execution of the analysis. Although our approach is substantially faster than mutation testing with traditional mutation operators, it is still conducted as a mutation analysis. Hence, a test suite's test cases will be executed multiple times during the analysis. Furthermore, our approach will not be applicable in scenarios in which traditional mutation testing is not applicable due to technical incompatibilities. This means that it will also be affected when, for example, an application uses special class-loading mechanisms that interfere with the byte-code replacement to put in place a mutation. However, this limitation can be circumvented by using the proposed machine-learning model to predict pseudo-tested methods. It operates on measures, which can be collected statically or in a single execution of a test suite. Future work is needed to determine whether the minimal stack distance measure is also computable in static analysis.

The machine-learning models for pseudo-tested methods, like those for low-fault-risk methods, are prediction models, which are inherently imperfect. Consequently, results contain false positives and miss false negatives. This limitation needs to be kept in mind when working with predicted outcomes of such models.

Limitations regarding the enhanced test case selection and prioritization approach include the fact that test cases may not be executable in the proposed order due to implicit dependencies between test cases or restrictions of the test framework. Furthermore, test cases may also fail due to changes to non-code artifacts (e.g., configuration files), which are not tracked in the current approach. In addition, it is necessary to ensure that test effectiveness information is collected efficiently and updated as seldom as reasonable, to avoid jeopardizing the gained benefits in reducing test efforts.

### 9.3 Future Work

Future work that could be undertaken based on presented techniques and gained insights in this dissertation is discussed below.

*Pseudo-testedness prediction in continuous integration pipeline:* An important topic for future work is the integration of the minimal stack distance computation and the prediction of pseudo-tested methods using the devised machine-learning models into a continuous integration pipeline. The execution of predictions within a continuous build would help provide developers with timely feedback on the testing state of recently changed methods. It will further enable researchers to better assess developers' acceptance and handling of pseudo-tested methods in real projects.

*Static computation of the minimal stack distance:* It is an open question whether the minimal stack distance measure to characterize the distance between a method and a test case can be approximated using a static code analysis. It would be interesting to evaluate the potential accuracy of such an approximation and whether the outcome is precise enough for use in machine-learning models to predict pseudo-tested methods. A static computation would offer speed advantages over the proposed dynamic analysis.

*Further improvement of prediction models for pseudo-tested methods:* We devised machine-learning models to predict pseudo-tested methods based



on method and test-case measures. A possible direction for future work is to improve the performance of these models by considering additional measures. It could be beneficial to include, for example, measures that quantify and characterize the use of assertions in test cases, as well as information about validations and checks within methods of the application code. In addition, cross-project prediction models could be enhanced by incorporating project characteristics into the models.

*Evaluation of incremental mutation analysis:* PIT provides an experimental feature to conduct an incremental mutation analysis (Coles, 2019b). It keeps results from a mutation analysis and uses them in future analyses on newer versions of a software to infer the mutation testing verdict of mutants that belong to unchanged code. According to Coles (2019b), the implementation makes assumptions and optimizations that cause results to become imprecise, to a certain degree. This feature has not been empirically evaluated yet. Future work is required in order to quantify and better understand the impact of the undertaken simplifications on the results, shed light on the achievable speed improvements compared to a full analysis, and give orientations to further improvements. Depending on the outcome, an incremental analysis could be used to determine pseudo-testedness in a continuous integration pipeline (instead of applying a prediction model).

*Multi-stage mutation testing process:* While pseudo-tested methods constitute ineffectively tested code, it cannot be inferred that the remaining methods are effectively tested. Therefore, another interesting path for future work is to set-up and evaluate a multi-stage mutation testing process. In a first step, the developed machine-learning model should be employed to identify methods that are pseudo-tested with high certainty. This implies that the model should be optimized towards precision. In a second step, mutation testing with the pseudo-testedness operator should be applied on methods for which the model cannot reliably predict the mutation testing verdict; this would identify further pseudo-tested methods. Then, in a third step, a mutation analysis using traditional operators could be employed

on non-pseudo-tested methods to determine their testing state at a more fine-grained level. Such a multi-stage process would enable an efficient detection of pseudo-tested methods and a deeper analysis of methods that have already been tested to a minimum degree. Any of the steps could be executed in an incremental analysis.

*Exclusion of unnecessary code in addition to low-fault-risk methods:* In [Haas et al. \(2019a\)](#) and [Haas et al. \(2019b\)](#), we proposed and evaluated a static analysis approach to identify unnecessary code at the file level. This approach is based on the hypothesis that the most stable and, at the same time, least central code in the dependency structure of a system tends to be unnecessary. The results of that study showed that recommendations of such an approach can give relevant pointers to unnecessary code. We suggest combining such an analysis with the identification of low-fault-risk methods to further restrict the relevant code to be tested. It would be interesting to empirically study on long-grown systems whether this can help developers and testers devote their testing efforts towards the relevant, actively used parts a software system.

# BIBLIOGRAPHY

- Acree Jr., A. T. (1980).** *On Mutation*. Tech. rep. Georgia Institute of Tech (cited on p. 49).
- Acree, A. T., T. A. Budd, R. A. DeMillo, R. J. Lipton, F. G. Sayward (1979).** *Mutation Analysis*. Tech. rep. Georgia Institute of Tech (cited on p. 62).
- Agrawal, H., J. R. Horgan, E. W. Krauser, S. A. London (1993a).** “Incremental Regression Testing.” In: *Proceedings of the 6th Conference on Software Maintenance (CSM’93)*. IEEE Computer Society (cited on p. 28).
- Agrawal, R., R. Srikant (1994).** “Fast Algorithms for Mining Association Rules.” In: *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB’94)*. Vol. 1215. Morgan Kaufmann Publishers (cited on p. 143).
- Agrawal, R., T. Imieliński, A. Swami (1993b).** “Mining Association Rules between Sets of Items in Large Databases.” In: *SIGMOD Record* 22.2. ACM (cited on p. 39).
- Ahmed, I., R. Gopinath, C. Brindescu, A. Groce, C. Jensen (2016).** “Can Testability Be Effectively Measured?” In: *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE’16)*. ACM (cited on p. 34, 44, 112).
- Ahmed, I., C. Jensen, A. Groce, P. E. McKenney (2017).** “Applying Mutation Analysis on Kernel Test Suites: An Experience Report.” In: *Proceedings of the 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW’17)*. IEEE (cited on p. 52).

- Andrews, J. H., L. C. Briand, Y. Labiche (2005).** “Is Mutation an Appropriate Tool for Testing Experiments?” In: *Proceedings of the 27th International Conference on Software Engineering (ICSE’05)* (cited on p. 32, 48).
- Andrews, J. H., L. C. Briand, Y. Labiche, A. S. Namin (2006).** “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria.” In: *Transactions on Software Engineering (TSE)* 32. IEEE (cited on p. 42, 48).
- Antinyan, V., J. Derehag, A. Sandberg, M. Staron (2018).** “Mythical Unit Test Coverage.” In: *Software* 35.3. IEEE (cited on p. 14, 44).
- Antoniol, G., K. Ayari, M. Di Penta, F. Khomh, Y.-G. Guéhéneuc (2008).** “Is It a Bug or An Enhancement?: A Text-Based Approach to Classify Change Requests.” In: *Proceedings of the 18th International Conference on Computer Science and Software Engineering (CASCON’08)*. Vol. 8. IBM (cited on p. 154).
- Apache Software Foundation (ASF) (2019).** *Introduction to the Build Lifecycle*. URL: <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html> (visited on May 16, 2019) (cited on p. 25).
- Arlot, S., A. Celisse (2010).** “A Survey of Cross-Validation Procedures for Model Selection.” In: *Statistics Surveys* 4. The American Statistical Association, the Bernoulli Society, the Institute of Mathematical Statistics, and the Statistical Society of Canada (cited on p. 36).
- Bacchelli, A., M. D’Ambros, M. Lanza (2010).** “Are Popular Classes More Defect Prone?” In: *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE’10)*. Springer (cited on p. 55).
- Bach, T., A. Andrzejak, R. Pannemans, D. Lo (2017).** “The Impact of Coverage on Bug Density in a Large Industrial Software Project.” In: *Proceedings of the 10th International Symposium on Empirical Software Engineering and Measurement (ESEM’17)*. IEEE (cited on p. 43).
- Bachmann, A., C. Bird, F. Rahman, P. Devanbu, A. Bernstein (2010).** “The Missing Links: Bugs and Bug-Fix Commits.” In: *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE’10)*. ACM (cited on p. 154).
- Bajada, L., M. Micallef, C. Colombo (2015).** “Using Control Flow Analysis to Improve the Effectiveness of Incremental Mutation Testing.” In: *Proceedings of the*

*14th International Workshop on Principles of Software Evolution (IWPSE'15)*. ACM (cited on p. 49).

**Baldwin, D., F. Sayward (1979)**. *Heuristics for Determining Equivalence of Program Mutations*. Tech. rep. Georgia Institute of Tech (cited on p. 51).

**Barr, E. T., M. Harman, P. McMinn, M. Shahbaz, S. Yoo (2015)**. “The Oracle Problem in Software Testing: A Survey.” In: *Transactions on Software Engineering (TSE)* 41.5 (cited on p. 23).

**Bayardo, R. J., R. Agrawal, D. Gunopulos (1999)**. “Constraint-Based Rule Mining in Large, Dense Databases.” In: *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*. IEEE (cited on p. 39).

**Beizer, B. (1990)**. *Software Testing Techniques (2nd Edition)*. Van Nostrand Reinhold Co. (cited on p. 25).

**Benoit, H. (2014)**. *Code Coverage: A misleading metric*. URL: <https://benohead.com/code-coverage-a-misleading-metric> (visited on June 1, 2017) (cited on p. 44).

**Berner, S., R. Weber, R. K. Keller (2005)**. “Observations and Lessons Learned from Automated Testing.” In: *Proceedings of the 27th International Conference on Software Engineering: Experience Reports (ICSE'05 ER)*. ACM (cited on p. 25).

**Bertolino, A. (2007)**. “Software Testing Research: Achievements, Challenges, Dreams.” In: *Proceedings of the Future of Software Engineering (FOSE'07)*. IEEE Computer Society (cited on p. 128).

**Besenreuther, T. (2014)**. “Dynamic Analysis of Test Quality.” MA thesis. Munich, Germany: Technische Universität München (cited on p. 15, 45, 46).

**Bird, C., A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, P. Devanbu (2009)**. “Fair and Balanced? Bias in Bug-Fix Datasets.” In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*. ACM (cited on p. 154).

**Bird, C., N. Nagappan, B. Murphy, H. Gall, P. Devanbu (2011)**. “Don't Touch My Code! Examining the Effects of Ownership on Software Quality.” In: *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the*

- Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*. ACM (cited on p. 55).
- Bogacki, B., B. Walter (2006)**. “Evaluation of Test Code Quality with Aspect-Oriented Mutations.” In: *Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP'06)*. Springer (cited on p. 49).
- Bowes, D., T. Hall, M. Harman, Y. Jia, F. Sarro, F. Wu (2016)**. “Mutation-Aware Fault Prediction.” In: *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM (cited on p. 55).
- Breiman, L. (2001)**. “Random Forests.” In: *Machine Learning* 45.1. Kluwer Academic Publishers (cited on p. 38, 39).
- Briand, L., D. Pfahl (1999)**. “Using Simulation for Assessing the Real Impact of Test Coverage on Defect Coverage.” In: *Proceedings 10th International Symposium on Software Reliability Engineering (ISSRE'99)*. IEEE (cited on p. 43).
- Brin, S., R. Motwani, C. Silverstein (1997)**. “Beyond Market Baskets: Generalizing Association Rules to Correlations.” In: *SIGMOD Record* 26.2. ACM (cited on p. 40).
- Brooks Jr., F. P. (1995)**. *The Mythical Man-Month (Anniversary Edition)*. Addison-Wesley (cited on p. 14).
- Buckland, M., F. Gey (1994)**. “The Relationship between Recall and Precision.” In: *Journal of the American Society for Information Science* 45.1. Wiley Online Library (cited on p. 36).
- Budd, T. A., D. Angluin (1982)**. “Two Notions of Correctness and Their Relation to Testing.” In: *Acta Informatica* 18.1. Springer (cited on p. 32, 51, 63).
- Burns, C. (2019)**. *What is Code Coverage and Why It Should Not Lead Development*. URL: <https://capgemini.github.io/testing/What-Is-Code-Coverage-and-Why-It-Should-Not-Lead-Development> (visited on June 1, 2017) (cited on p. 44).
- Bytes, J. (2016)**. *Code Coverage Should Not Be The Goal*. URL: <https://jeremybytes.blogspot.com/2016/08/code-coverage-should-not-be-goal.html> (visited on June 1, 2017) (cited on p. 44).

- Cachia, M. A., M. Micallef, C. Colombo (2013).** “Towards Incremental Mutation Testing.” In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 294. Elsevier (cited on p. 49).
- Carvalho, L., M. A. Guimarães, M. Ribeiro, L. Fernandes, M. Al-Hajjaji, R. Gheyi, T. Thüm (2018).** “Equivalent Mutants in Configurable Systems: An Empirical Study.” In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS’19)*. ACM (cited on p. 66).
- Catal, C. (2011).** “Software Fault prediction: A Literature Review and Current Trends.” In: *Expert Systems with Applications* 38.4. Elsevier (cited on p. 128).
- Catal, C., B. Diri (2009).** “A Systematic Review of Software Fault Prediction Studies.” In: *Expert Systems with Applications* 36.4. Elsevier (cited on p. 54).
- Catal, C., D. Mishra (2013).** “Test Case Prioritization: A Systematic Mapping Study.” In: *Software Quality Journal (SQJ)* 21.3. Springer (cited on p. 57).
- Chan, R. (2019).** *The 10 Most Popular Programming Languages, According to the ‘Facebook for Programmers’*. URL: <https://www.businessinsider.de/the-10-most-popular-programming-languages-according-to-github-2018-10> (visited on July 5, 2019) (cited on p. 17).
- Chawla, N. V., K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer (2002).** “SMOTE: Synthetic Minority Over-Sampling Technique.” In: *Journal of Artificial Intelligence Research (JAIR)* 16 (cited on p. 118, 136).
- Chekam, T. T., M. Papadakis, Y. Le Traon, M. Harman (2017).** “An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption.” In: *Proceedings of the 39th International Conference on Software Engineering (ICSE’17)*. IEEE (cited on p. 48).
- Chen, M.-H., A. P. Mathur, V. J. Rego (1995).** “Effect of Testing Techniques on Software Reliability Estimates Obtained Using a Time-Domain Model.” In: *Transactions on Reliability* 44.1. IEEE (cited on p. 42).
- Chilenski, J. J., S. P. Miller (1994).** “Applicability of Modified Condition/Decision Coverage to Software Testing.” In: *Software Engineering Journal* 9.5. IET (cited on p. 14).

- Cohn, M. (2010).** *Succeeding with Agile: Software Development using Scrum*. Pearson Education (cited on p. 25, 26).
- Coles, H. (2019a).** *Pitest: FAQ*. URL: <http://pitest.org/faq> (visited on May 13, 2019) (cited on p. 34).
- **(2019b).** *Pitest: Incremental Analysis*. URL: [http://pitest.org/incremental\\_analysis](http://pitest.org/incremental_analysis) (visited on May 13, 2019) (cited on p. 35, 177).
- **(2019c).** *Pitest: Mutators Overview*. URL: <http://pitest.org/quickstart/mutators> (visited on May 18, 2018) (cited on p. 35).
- Coles, H., T. Laurent, C. Henard, M. Papadakis, A. Ventresque (2016).** “Demo: PIT: A Practical Mutation Testing Tool For Java.” In: *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA’16)*. ACM (cited on p. 34, 35, 63, 112).
- Czibula, G., Z. Marian, I. G. Czibula (2014).** “Software Defect Prediction Using Relational Association Rule Mining.” In: *Information Sciences 264*. Elsevier (cited on p. 40).
- D’Ambros, M., M. Lanza, R. Robbes (2012).** “Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison.” In: *Empirical Software Engineering 17.4-5*. Springer (cited on p. 55, 128, 154).
- Daran, M., P. Thévenod-Fosse (1996).** “Software Error Analysis: A Real Case Study Involving Real Faults and Mutations.” In: *SIGSOFT Software Engineering Notes (SEN) 21*. ACM (cited on p. 48).
- DeMillo, R. A., R. J. Lipton, F. G. Sayward (1978).** “Hints on Test Data Selection: Help for the Practicing Programmer.” In: *Computer 11.4*. IEEE (cited on p. 30, 31, 59).
- Del Frate, F., P. Garg, A. P. Mathur, A. Pasquini (1995).** “On the Correlation Between Code Coverage and Software Reliability.” In: *Proceedings of 6th International Symposium on Software Reliability Engineering (ISSRE’95)*. IEEE (cited on p. 42).
- Delahaye, M., L. Du Bousquet (2013).** “A Comparison of Mutation Analysis Tools for Java.” In: *Proceedings of the 13th International Conference on Quality Software (QSIC’13)*. IEEE (cited on p. 34, 51, 63).



- Delplanque, J., S. Ducasse, G. Polito, A. Black, A. Etien (2019).** “Rotten Green Tests.” In: *Proceedings of the 41st International Conference on Software Engineering (ICSE’19)* (cited on p. 54).
- Diakopoulos, N., S. Cass (2016).** *Interactive: The Top Programming Languages 2016*. IEEE Spectrum. URL: <http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016> (visited on July 5, 2019) (cited on p. 17).
- Dietterich, T. G. (2000).** “Ensemble Methods in Machine Learning.” In: *Proceedings of the 1st International Workshop on Multiple Classifier Systems (MCS’00)*. Springer (cited on p. 38).
- Dreier, F. (2017).** “Obtaining Coverage per Test Case.” MA thesis. Munich, Germany: Technische Universität München (cited on p. 164, 166).
- Dustin, E., T. Garrett, B. Gauf (2009).** *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Pearson Education (cited on p. 14).
- Duvall, P. M., S. Matyas, A. Glover (2007).** *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education (cited on p. 162, 163).
- Eder, S., B. Hauptmann, M. Junker, E. Juergens, R. Vaas, K.-H. Prommer (2013).** “Did We Test Our Changes? Assessing Alignment Between Tests and Development in Practice.” In: *Proceedings of the 8th International Workshop on Automation of Software Test (AST’13)*. IEEE (cited on p. 158).
- Elbaum, S., A. G. Malishevsky, G. Rothermel (2002).** “Test Case Prioritization: A Family of Empirical Studies.” In: *Transactions on Software Engineering (TSE)* 28.2. IEEE (cited on p. 57, 58).
- Elbaum, S., H. N. Chin, M. B. Dwyer, J. Dokulil (2006).** “Carving Differential Unit Test Cases from System Test Cases.” In: *Proceedings of the 14th International Symposium on Foundations of Software Engineering (FSE’06)* (cited on p. 24, 25).
- Engström, E., P. Runeson, M. Skoglund (2010).** “A Systematic Review on Regression Test Selection Techniques.” In: *Information and Software Technology (IST)* 52.1. Elsevier (cited on p. 57).

- Farooq, F., A. Nadeem (2017).** “A Fault Based Approach to Test Case Prioritization.” In: *Proceedings of the 6th International Conference on Frontiers of Information Technology (FIT’17)*. IEEE (cited on p. 58).
- Feller, W (1968).** *An Introduction to Probability Theory and its Applications*. Vol. 1. John Wiley & Sons (cited on p. 38).
- Fleyshgakker, V. N., S. N. Weiss (1994).** “Efficient Mutation Analysis: A New Approach.” In: *Proceedings of the 3rd International Symposium on Software Testing and Analysis (ISSTA’94)*. ACM (cited on p. 49).
- Fowler, M. (2004).** *AssertionFreeTesting*. URL: <http://martinfowler.com/bliki/AssertionFreeTesting.html> (visited on Jan. 26, 2017) (cited on p. 16, 22).
- **(2012a).** *TestCoverage*. URL: <https://martinfowler.com/bliki/TestCoverage.html> (visited on June 1, 2017) (cited on p. 44).
  - **(2012b).** *TestPyramid*. URL: <https://martinfowler.com/bliki/TestPyramid.html> (visited on May 2, 2019) (cited on p. 25, 26).
- Frankl, P. G., O. Iakounenko (1998).** “Further Empirical Studies of Test Effectiveness.” In: *SIGSOFT Software Engineering Notes (SEN)* 23.6. ACM (cited on p. 42).
- Frankl, P. G., S. N. Weiss, C. Hu (1997).** “All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness.” In: *Journal of Systems and Software (JSS)* 38.3. Elsevier (cited on p. 32, 42, 48).
- Fraser, G., A. Arcuri (2011).** “EvoSuite: Automatic Test Suite Generation for Object-Oriented Software.” In: *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE’11)*. ACM (cited on p. 22).
- Giger, E., M. D’Ambros, M. Pinzger, H. C. Gall (2012).** “Method-Level Bug Prediction.” In: *Proceedings of the 6th International Symposium on Empirical Software Engineering and Measurement (ESEM’12)*. ACM (cited on p. 55, 134, 135, 138, 145, 146, 152–154).
- Gligoric, M., A. Groce, C. Zhang, R. Sharma, M. A. Alipour, D. Marinov (2013).** “Comparing Non-Adequate Test Suites using Coverage Criteria.” In: *Proceedings*

of the 22nd International Symposium on Software Testing and Analysis (ISSTA'13). ACM (cited on p. 43).

- (2015). “Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites.” In: *Transactions on Software Engineering and Methodology (TOSEM)* 24.4. ACM (cited on p. 43).

**Gonzalez-Hernandez, L., B. Lindström, J. Offutt, S. F. Andler, P. Potena, M. Bohlin (2018).** “Using Mutant Stubbornness to Create Minimal and Prioritized Test Sets.” In: *Proceedings of the 18th International Conference on Software Quality, Reliability and Security (QRS'18)*. IEEE (cited on p. 58).

**Gopinath, R., C. Jensen, A. Groce (2014a).** “Code Coverage for Suite Evaluation by Developers.” In: *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM (cited on p. 43, 48).

- (2014b). *Mutant Census: An Empirical Examination of the Competent Programmer Hypothesis*. Tech. rep. Oregon State University, Department of Computer Science (cited on p. 30).

**Gopinath, R., A. Alipour, I. Ahmed, C. Jensen, A. Groce (2015).** “How Hard Does Mutation Analysis Have to Be, Anyway?” In: *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE'15)*. IEEE (cited on p. 34, 112).

**Gopinath, R., M. A. Alipour, I. Ahmed, C. Jensen, A. Groce (2016).** “On the Limits of Mutation Reduction Strategies.” In: *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. IEEE (cited on p. 34, 112).

**Gopinath, R., I. Ahmed, M. A. Alipour, C. Jensen, A. Groce (2017).** “Does Choice of Mutation Tool Matter?” In: *Software Quality Journal (SQJ)* 25.3. Springer (cited on p. 34, 63).

**Gosling, J., B. Joy, G. Steele, G. Bracha, A. Buckley (2013).** *The Java Language Specification, Java SE 7 Edition, February 2012*. URL: <http://docs.oracle.com/javase/specs/jls/se7/html/index.html> (visited on Apr. 4, 2019) (cited on p. 17, 133).

- (2015). *The Java Language Specification (Java SE 8 Edition)*. Oracle America, Inc. (cited on p. 71).

- Graves, T. L., M. J. Harrold, J.-M. Kim, A. Porter, G. Rothermel (2001).** “An Empirical Study of Regression Test Selection Techniques.” In: *Transactions on Software Engineering and Methodology (TOSEM)* 10.2. ACM (cited on p. 56).
- Green, S. D., D. Kostovarov (2013).** *Test Assertions Guidelines Version 1.0*. OASIS Open (cited on p. 23).
- Grün, B. J., D. Schuler, A. Zeller (2009).** “The Impact of Equivalent Mutants.” In: *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW’09)*. IEEE (cited on p. 33, 51, 64, 123).
- Haas, R., R. Niedermayr, T. Roehm, S. Apel (2019a).** “Is Static Analysis Able to Identify Unnecessary Source Code?” In: *Transactions on Software Engineering and Methodology (TOSEM)*. To appear. ACM (cited on p. 178).
- **(2019b).** “Poster: Recommending Unnecessary Source Code Based on Static Analysis.” In: *Proceedings of the 41st International Conference on Software Engineering Companion (ICSE’19 Companion)*. IEEE (cited on p. 178).
- Haas, R., R. Niedermayr, E. Juergens (2019c).** “Teamscale: Tackle Technical Debt and Control the Quality of Your Software.” In: *Proceedings of the 2nd International Conference on Technical Debt (TechDebt’19 Tools Track)*. IEEE (cited on p. 108, 135, 163).
- Hahsler, M., B. Gruen, K. Hornik (2005).** “arules – A Computational Environment for Mining Association Rules and Frequent Item Sets.” In: *Journal of Statistical Software (JSS)* 14.15 (cited on p. 143).
- Hahsler, M., C. Buchta, B. Gruen, K. Hornik (2018).** *arules: Mining Association Rules and Frequent Itemsets*. R package version 1.6-1. URL: <https://CRAN.R-project.org/package=arules> (cited on p. 143).
- Hall, T., S. Beecham, D. Bowes, D. Gray, S. Counsell (2012).** “A Systematic Literature Review on Fault Prediction Performance in Software Engineering.” In: *Transactions on Software Engineering (TSE)* 38.6. IEEE (cited on p. 54, 128).
- Hamill, P. (2004).** *Unit Test Frameworks: Tools for High-Quality Software Development*. O’Reilly Media (cited on p. 26).

- Harman, M., R. Hierons, S. Danicic (2001).** “The Relationship Between Program Dependence and Mutation Analysis.” In: *Mutation Testing for the New Century*. Springer (cited on p. 51).
- Hassan, A. E. (2009).** “Predicting Faults Using the Complexity of Code Changes.” In: *Proceedings of the 31st International Conference on Software Engineering (ICSE’09)*. IEEE Computer Society (cited on p. 55).
- Hata, H., O. Mizuno, T. Kikuno (2012).** “Bug Prediction Based on Fine-Grained Module Histories.” In: *Proceedings of the 34th International Conference on Software Engineering (ICSE’12)*. IEEE (cited on p. 55).
- Hayhurst, K. J., D. S. Veerhusen, J. J. Chilenski, L. K. Rierson (2001).** *A Practical Tutorial on Modified Condition/Decision Coverage*. Tech. rep. TM-2001-210876. NASA (cited on p. 29).
- He, Z., F. Shu, Y. Yang, M. Li, Q. Wang (2012).** “An Investigation on the Feasibility of Cross-Project Defect Prediction.” In: *Automated Software Engineering* 19.2. Springer (cited on p. 55, 128, 137, 159).
- Hemmati, H. (2015).** “How Effective Are Code Coverage Criteria?” In: *Proceedings of the 15th International Conference on Software Quality, Reliability and Security (QRS’15)*. IEEE (cited on p. 44, 64, 108).
- Herzig, K., S. Just, A. Zeller (2013).** “It’s Not a Bug, It’s a Feature: How Misclassification Impacts Bug Prediction.” In: *Proceedings of the 35th International Conference on Software Engineering (ICSE’13)*. IEEE (cited on p. 154).
- Hierons, R., M. Harman, S. Danicic (1999).** “Using Program Slicing to Assist in the Detection of Equivalent Mutants.” In: *Software Testing, Verification and Reliability (STVR)* 9.4. Wiley Online Library (cited on p. 51).
- Hilton, M., N. Nelson, T. Tunnell, D. Marinov, D. Dig (2017).** “Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility.” In: *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE’17)*. ACM (cited on p. 17).

- Hilton, M., J. Bell, D. Marinov (2018).** “A Large-Scale Study of Test Coverage Evolution.” In: *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE’18)* (cited on p. 66).
- Homès, B. (2013).** *Fundamentals of Software Testing*. John Wiley & Sons (cited on p. 22, 29).
- Horgan, J. R., S. London, M. R. Lyu (1994).** “Achieving Software Quality with Testing Coverage Measures.” In: *Computer* 27.9. IEEE (cited on p. 42).
- Howden, W. E. (1982).** “Weak Mutation Testing and Completeness of Test Sets.” In: *Transactions on Software Engineering (TSE)* 4. IEEE (cited on p. 49).
- Huang, J. (1975).** “An Approach to Program Testing.” In: *Computing Surveys (CSUR)* 7.3. ACM (cited on p. 14).
- Hummel, B. (2014).** *McCabe’s Cyclomatic Complexity and Why We Don’t Use It*. URL: <https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity> (visited on Mar. 4, 2019) (cited on p. 130).
- Hutchins, M., H. Foster, T. Goradia, T. Ostrand (1994).** “Experiments on the Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria.” In: *Proceedings of the 16th International Conference on Software Engineering (ICSE’94)* (cited on p. 42, 48).
- Inozemtseva, L., R. Holmes (2014).** “Coverage Is Not Strongly Correlated With Test Suite Effectiveness.” In: *Proceedings of the 36th International Conference on Software Engineering (ICSE’14)*. ACM (cited on p. 14, 44, 102, 108).
- Institute of Electrical and Electronics Engineers (IEEE) (2010).** *IEEE Standard Classification for Software Anomalies*. Tech. rep. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) (cited on p. 28).
- Ivanković, G. P. M., B. Kurtz, P. Ammann, R. Just (2018).** “An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions.” In: *Proceedings of the 13th International Workshop on Mutation Analysis (MUTATION’18)* (cited on p. 15, 32, 33, 53, 64).

- Jalbert, K., J. S. Bradbury (2012).** “Predicting Mutation Score using Source Code and Test Suite Metrics.” In: *Proceedings of the 1st International Workshop on Realizing AI Synergies in Software Engineering (RAISE’12)*. IEEE (cited on p. 50).
- Ji, C., Z. Chen, B. Xu, Z. Zhao (2009).** “A Novel Method of Mutation Clustering Based on Domain Analysis.” In: *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE’09)*. Vol. 9 (cited on p. 49).
- Jia, Y., M. Harman (2008).** “Constructing Subtle Faults Using Higher Order Mutation Testing.” In: *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM’08)*. IEEE (cited on p. 49).
- **(2011).** “An Analysis and Survey of the Development of Mutation Testing.” In: *Transactions on Software Engineering (TSE)* 37.5. IEEE (cited on p. 15, 30–32, 49, 64, 102).
- Jin, Z., A. J. Offutt (1998).** “Coupling-Based Criteria for Integration Testing.” In: *Software Testing, Verification and Reliability (STVR)* 8.3. Wiley Online Library (cited on p. 24, 25).
- Juergens, E., D. Pagano (2016).** *Did We Test the Right Thing? Experiences with Test Gap Analysis in Practice*. Whitepaper. CQSE GmbH (cited on p. 157).
- Just, R., D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser (2014a).** “Are Mutants a Valid Substitute for Real Faults in Software Testing?” In: *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE’14)*. ACM (cited on p. 32, 48).
- Just, R., D. Jalali, M. D. Ernst (2014b).** “Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs.” In: *Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA’14)*. ACM (cited on p. 55, 108, 129, 130, 138, 154).
- Khoshgoftaar, T. M., K. Gao, N. Seliya (2010).** “Attribute Selection and Imbalanced Data: Problems in Software Defect Prediction.” In: *Proceedings of the 22nd International Conference on Tools with Artificial Intelligence (ICTAI’10)*. Vol. 1. IEEE (cited on p. 135).

- Kim, S., T. Zimmermann, E. J. Whitehead Jr., A. Zeller (2007).** “Predicting Faults From Cached History.” In: *Proceedings of the 29th International Conference on Software Engineering (ICSE’07)*. IEEE Computer Society (cited on p. 55).
- Kim, S., E. J. Whitehead Jr., Y. Zhang (2008).** “Classifying Software Changes: Clean or Buggy?” In: *Transactions on Software Engineering (TSE)* 34.2. IEEE (cited on p. 55).
- Kim, S., H. Zhang, R. Wu, L. Gong (2011).** “Dealing with Noise in Defect Prediction.” In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE’11)*. IEEE (cited on p. 154).
- King, K. N., A. J. Offutt (1991).** “A Fortran Language System for Mutation-Based Software Testing.” In: *Software: Practice and Experience (SPE)* 21.7. Wiley Online Library (cited on p. 31, 62).
- Kintis, M., M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, M. Harman (2017).** “Detecting Trivial Mutant Equivalences via Compiler Optimisations.” In: *Transactions on Software Engineering (TSE)* 44.4. IEEE (cited on p. 51).
- Klischies, D., K. Fögen (2016).** “An Analysis of Current Mutation Testing Techniques Applied to Real World Examples.” In: *Full-scale Software Engineering / Current Trends in Release Engineering* 13. RWTH Aachen University, Research Group Software Construction (cited on p. 52).
- Kobayashi, H., B. L. Mark, W. Turin (2011).** *Probability, Random Processes, and Statistical Analysis: Applications to Communications, Signal Processing, Queueing Theory and Mathematical Finance*. Cambridge University Press (cited on p. 36).
- Kochhar, P. S., F. Thung, D. Lo (2015).** “Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems.” In: *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER’15)*. IEEE (cited on p. 43).
- Kochhar, P. S., D. Lo, J. Lawall, N. Nagappan (2017).** “Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects.” In: *Transactions on Reliability* 66.4. IEEE (cited on p. 44).
- Kohavi, R. et al. (1995).** “A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection.” In: *Proceedings of the 14th International Joint*



*Conferences on Artificial Intelligence Organization (IJCAI'95)*. Vol. 14. 2 (cited on p. 111).

**Kuhn, M., J. Wing, S. Weston, A. Williams, C. Keefer, A. Engelhardt, T. Cooper, Z. Mayer, B. Kenkel, M. Benesty, R. Lescarbeau, A. Ziem, L. Scrucca, Y. Tang, C. Candan, T. Hunt, the R Core Team (2017)**. *caret: Classification and Regression Training*. R package version 6.0-76. URL: <https://CRAN.R-project.org/package=caret> (cited on p. 113, 142).

**Landis, J. R., G. G. Koch (1977)**. “The Measurement of Observer Agreement for Categorical Data.” In: *Biometrics*. Biometric Society (cited on p. 92).

**Langr, J., A. Hunt, D. Thomas (2015)**. *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf (cited on p. 24).

**Lee, N. (2016)**. *Why Test Code Coverage Targets are a Bad Idea*. URL: <https://medium.com/@nicklee1/why-test-code-coverage-targets-are-a-bad-idea-1b9b8ef711ef> (visited on June 1, 2017) (cited on p. 44).

**Lee, T., J. Nam, D. Han, S. Kim, H. P. In (2011)**. “Micro Interaction Metrics for Defect Prediction.” In: *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*. ACM (cited on p. 55).

**Li, N., J. Offutt (2017)**. “Test Oracle Strategies for Model-Based Testing.” In: *Transactions on Software Engineering (TSE)* 43.4. IEEE (cited on p. 23, 25, 28).

**Li, N., U. Praphamontripong, J. Offutt (2009)**. “An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage.” In: *Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'09)*. IEEE (cited on p. 48).

**Liaw, A., M. Wiener (Dec. 2002)**. “Classification and Regression by RandomForest.” In: *R News* 2.3 (cited on p. 38, 39).

**Lipton, R. J. (1971)**. *Fault Diagnosis of Computer Programs*. Tech. rep. Carnegie Mellon University (cited on p. 30).

- Longadge, R., S. Dongre, L. Malik (2013).** “Class Imbalance Problem in Data Mining: Review.” In: *International Journal of Computer Science and Network (IJCSN)* 2.1. Digital Library of Academic Research (cited on p. 136).
- Lou, Y., D. Hao, L. Zhang (2015).** “Mutation-Based Test-Case Prioritization in Software Evolution.” In: *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE'15)*. IEEE (cited on p. 58).
- Ma, B., K. Dejaeger, J. Vanthienen, B. Baesens (2010).** “Software Defect Prediction Based on Association Rule Classification.” In: *Proceedings of the 1st International Conference on E-Business Intelligence (ICEBI'10)*. Atlantis (cited on p. 40).
- Ma, Y.-S., J. Offutt, Y. R. Kwon (2005).** “MuJava: An Automated Class Mutation System.” In: *Software Testing, Verification and Reliability (STVR)* 15.2. Wiley Online Library (cited on p. 49).
- Madeyski, L. (2009).** *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer Science & Business Media (cited on p. 162).
- Madeyski, L., M. Jureczko (2015).** “Which Process Metrics can Significantly Improve Defect Prediction Models? An Empirical Study.” In: *Software Quality Journal (SQJ)* 23.3. Springer (cited on p. 155).
- Madeyski, L., W. Orzeszyna, R. Torkar, M. Jozala (2014).** “Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation.” In: *Transactions on Software Engineering (TSE)* 40.1. IEEE (cited on p. 15, 33, 58, 64).
- Mansoor, U. (2016).** *Do not Misuse Code Coverage*. URL: <https://codeahoy.com/2016/04/16/do-not-misuse-code-coverage> (visited on June 1, 2017) (cited on p. 44).
- Marick, B. (1997).** *How to Misuse Code Coverage*. URL: <http://www.exampler.com/testing-com/writings/coverage.pdf> (visited on Jan. 26, 2017) (cited on p. 44).
- Mathur, A. P. (2013).** *Foundations of Software Testing, second edition*. Pearson Education India (cited on p. 29).

- Mathur, A. P., W. E. Wong (1994).** “An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria.” In: *Software Testing, Verification and Reliability (STVR) 4.1*. Wiley Online Library (cited on p. 48).
- McCabe, T. J. (1976).** “A Complexity Measure.” In: *Transactions on Software Engineering (TSE) 4*. IEEE (cited on p. 130).
- McCallum, A., K. Nigam (1998).** “A Comparison of Event Models for Naive Bayes Text Classification.” In: *Proceedings of the Workshop on Learning for Text Categorization (AAAI-98-W7)*. Vol. 752. AIII (cited on p. 134).
- Mende, T., R. Koschke (2009).** “Revisiting the Evaluation of Defect Prediction Models.” In: *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE’09)*. ACM, p. 7 (cited on p. 135).
- Meneely, A., L. Williams, W. Snipes, J. Osborne (2008).** “Predicting Failures with Developer Networks and Social Network Analysis.” In: *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE’08)*. ACM (cited on p. 55).
- Menzies, T., J. S. Di Stefano (2004).** “How Good is your Blind Spot Sampling Policy?” In: *Proceedings of the 8th International Symposium on High Assurance Systems Engineering (HASE’04)*. IEEE (cited on p. 128).
- Menzies, T., J. S. Di Stefano, M. Chapman, K. McGill (2002).** “Metrics That Matter.” In: *Proceedings of the 27th NASA Goddard Software Engineering Workshop*. IEEE. IEEE/NASA (cited on p. 132).
- Menzies, T., J. Stefano, K. Ammar, K. McGill, P. Callis, J Davis, R Chapman (2003).** “When Can We Test Less?” In: *Proceedings of the 9th International Symposium on Software Metrics (SMS’03)*. IEEE (cited on p. 128).
- Menzies, T., J. DiStefano, A. Orrego, R Chapman (2004).** “Assessing Predictors of Software Defects.” In: *Proceedings of the Workshop Predictive Software Models (PSM’04)* (cited on p. 132).
- Menzies, T., J. Greenwald, A. Frank (2007).** “Data Mining Static Code Attributes to Learn Defect Predictors.” In: *Transactions on Software Engineering (TSE) 33.1*. IEEE (cited on p. 55, 128).

- Menzies, T., Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener (2010).** “Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches.” In: *Automated Software Engineering* 17.4. Springer (cited on p. 135).
- Migliani, G. (2017).** *Dynamic Method Dispatch or Runtime Polymorphism in Java*. URL: <https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java> (visited on Apr. 5, 2019) (cited on p. 66).
- Mockus, A., N. Nagappan, T. T. Dinh-Trong (2009).** “Test Coverage and Post-Verification Defects: A Multiple Case Study.” In: *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM’09)* (cited on p. 42).
- Mols, J. (2017).** *The 100% code coverage problem*. URL: <https://jeroenmols.com/blog/2017/11/28/coveragproblem> (visited on June 1, 2017) (cited on p. 44).
- Morell, L. J. (1984).** “A Theory of Error-Based Testing.” PhD thesis. Maryland, MD, USA: University of Maryland (cited on p. 28).
- Možucha, J., B. Rossi (2016).** “Is Mutation Testing Ready to Be Adopted Industry-Wide?” In: *Proceedings of the 17th International Conference on Product-Focused Software Process Improvement (PROFES’16)*. Springer (cited on p. 52).
- Muşlu, K., Y. Brun, M. D. Ernst, D. Notkin (2015).** “Reducing Feedback Delay of Software Development Tools via Continuous Analysis.” In: *Transactions on Software Engineering (TSE)* 41.8. IEEE (cited on p. 162).
- Nagappan, N., T. Ball (2005).** “Use of Relative Code Churn Measures to Predict System Defect Density.” In: *Proceedings of the 27th International Conference on Software Engineering (ICSE’05)*. IEEE (cited on p. 55).
- Nagappan, N., T. Ball, A. Zeller (2006).** “Mining Metrics to Predict Component Failures.” In: *Proceedings of the 28th International Conference on Software Engineering (ICSE’06)*. ACM (cited on p. 55).
- Naik, K., P. Tripathy (2008).** *Software Testing and Quality Assurance*. Wiley Online Library (cited on p. 23).

- Namin, A. S., J. H. Andrews (2009).** “The Influence of Size and Coverage on Test Suite Effectiveness.” In: *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)* (cited on p. 43).
- Namin, A. S., J. H. Andrews, D. J. Murdoch (2008).** “Sufficient Mutation Operators for Measuring Test Effectiveness.” In: *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM (cited on p. 49, 50).
- Neward, T. (2013).** “Java 8: Lambdas, Part 1.” In: *Java Magazine* 4. Oracle Inc. (Cited on p. 155).
- Nica, S., R. Ramler, F. Wotawa (2011).** “Is Mutation Testing Scalable for Real-World Software Projects?” In: *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID'11)* (cited on p. 51).
- Niedermayr, R. (2013).** “Meaningful and Practical Measures for Regression Test Reliability.” MA thesis. Munich, Germany: Technische Universität München (cited on p. 17, 64).
- (2018). *Pitest: Pull Request for Computing a Full Mutation Matrix*. Pull Request. URL: <https://github.com/hcoles/pitest/pull/511> (cited on p. 122).
- Niedermayr, R., S. Wagner (2019).** “Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?” In: *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering (EASE'19)*. ACM (cited on p. 103).
- Niedermayr, R., E. Juergens, S. Wagner (2016).** “Will My Tests Tell Me If I Break This Code?” In: *Proceedings of the 1st International Workshop on Continuous Software Evolution and Delivery (CSED'16)*. ACM (cited on p. 53, 64, 74, 78, 97, 123).
- Niedermayr, R., T. Röhm, S. Wagner (2018a).** *Dataset: Too Trivial To Test?* URL: [https://figshare.com/articles/Dataset\\_Too\\_Trivial\\_To\\_Test\\_/6194171](https://figshare.com/articles/Dataset_Too_Trivial_To_Test_/6194171) (cited on p. 140, 143, 145).
- (2018b). “Poster: Identification of Methods with Low Fault Risk.” In: *Proceedings of the 40th International Conference on Software Engineering Companion (ICSE'18 Companion)*. ACM (cited on p. 130).

- Niedermayr, R., T. Röhm, S. Wagner (2019).** “Too Trivial To Test? An Inverse View on Defect Prediction to Identify Methods with Low Fault Risk.” In: *PeerJ Computer Science* 5. PeerJ (cited on p. 130).
- Offutt, A. J. (1992).** “Investigations of the Software Testing Coupling Effect.” In: *Transactions on Software Engineering and Methodology (TOSEM)* 1.1. ACM (cited on p. 31).
- Offutt, A. J., J. Pan (1996).** “Detecting Equivalent Mutants and the Feasible Path Problem.” In: *Proceedings of 11th Annual Conference on Computer Assurance (COM-PASS’96)*. IEEE (cited on p. 51).
- **(1997).** “Automatically Detecting Equivalent Mutants and Infeasible Paths.” In: *Software Testing, Verification and Reliability (STVR)* 7.3. Wiley Online Library (cited on p. 33, 51).
- Offutt, A. J., R. H. Untch (2001).** “Mutation 2000: Uniting the Orthogonal.” In: *Mutation Testing for the New Century*. Springer (cited on p. 30, 48).
- Offutt, A. J., G. Rothermel, C. Zapf (1993).** “An Experimental Evaluation of Selective Mutation.” In: *Proceedings of the 15th International Conference on Software Engineering (ICSE’93)*. IEEE Computer Society (cited on p. 49).
- Offutt, A. J., A. Lee, G. Rothermel, R. H. Untch, C. Zapf (1996a).** “An Experimental Determination of Sufficient Mutant Operators.” In: *Transactions on Software Engineering and Methodology (TOSEM)* 5.2. ACM (cited on p. 32, 49).
- Offutt, A. J., J. Pan, K. Tewary, T. Zhang (1996b).** “An Experimental Evaluation of Data Flow and Mutation Testing.” In: *Software: Practice and Experience (SPE)* 26.2 (cited on p. 48).
- Offutt, J. (1988).** “Automatic Test Data Generation.” PhD thesis. Atlanta, GA, USA: Georgia Institute of Technology (cited on p. 28).
- Osherove, R. (2009).** *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co. (cited on p. 24).
- Ostrand, T. J., E. J. Weyuker, R. M. Bell (2004).** “Where the Bugs Are.” In: *SIGSOFT Software Engineering Notes (SEN)* 29.4. ACM (cited on p. 155).

- (2005). “Predicting the Location and Number of Faults in Large Software Systems.” In: *Transactions on Software Engineering (TSE)* 31.4. IEEE (cited on p. 128).
- Pal, M. (2005).** “Random Forest Classifier for Remote Sensing Classification.” In: *International Journal of Remote Sensing (IJRS)* 26.1. Taylor & Francis (cited on p. 38, 39).
- Palomba, F., M. Zanoni, F. A. Fontana, A. De Lucia, R. Oliveto (2016).** “Smells like Teen Spirit: Improving Bug Prediction Performance using the Intensity of Code Smells.” In: *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME’16)*. IEEE (cited on p. 55).
- Papadakis, M., Y. Jia, M. Harman, Y. Le Traon (2015).** “Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique.” In: *Proceedings of the 37th International Conference on Software Engineering (ICSE’15)*. IEEE (cited on p. 51, 66).
- Papadakis, M., M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman (2017).** “Mutation Testing Advances: An Analysis and Survey.” In: *Advances in Computers*. Elsevier (cited on p. 30).
- Papadakis, M., D. Shin, S. Yoo, D.-H. Bae (2018).** “Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship Between Mutants and Real Faults.” In: *Proceedings of the 40th International Conference on Software Engineering (ICSE’40)*. IEEE (cited on p. 48).
- Pascarella, L., F. Palomba, A. Bacchelli (2018).** “Re-Evaluating Method-Level Bug Prediction.” In: *Proceedings of the 25th International Conference on Software Analysis, Evolution, and Reengineering (SANER’18)*. IEEE (cited on p. 55).
- Petrović, G., M. Ivanković (2018).** “State of Mutation Testing at Google.” In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE’18 SEIP)*. ACM (cited on p. 33, 53, 99).
- Radio Technical Commission for Aeronautics (RTCA) (1992).** *DO-178B: Software Considerations in Airborne Systems and Equipment Certification* (cited on p. 29).
- Rahman, F., P. Devanbu (2011).** “Ownership, Experience and Effects: A Fine-Grained Study of Authorship.” In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE’11)*. ACM (cited on p. 55).

- Rajan, A., M. W. Whalen, M. P. Heimdahl (2008).** “The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage.” In: *Proceedings of the 30th International Conference on Software Engineering (ICSE’08)*. ACM (cited on p. 29, 43).
- Ramler, R., T. Wetzlmaier, C. Klammer (2017).** “An Empirical Study on the Application of Mutation Testing for a Safety-Critical Industrial Software System.” In: *Proceedings of the 32nd Symposium on Applied Computing (SAC’17)*. ACM (cited on p. 48, 52).
- Rapps, S., E. J. Weyuker (1982).** “Data Flow Analysis Techniques for Test Data Selection.” In: *Proceedings of the 6th International Conference on Software Engineering (ICSE’82)*. IEEE Computer Society (cited on p. 14).
- Rothermel, G., M. J. Harrold (1996).** “Analyzing Regression Test Selection Techniques.” In: *Transactions on Software Engineering (TSE)* 22.8. IEEE (cited on p. 56).
- Rothermel, G., M. J. Harrold, J. Ostrin, C. Hong (1998).** “An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites.” In: *Proceedings of the 14th International Conference on Software Maintenance (ICSM’98)*. IEEE (cited on p. 57).
- Rothermel, G., R. H. Untch, C. Chu, M. J. Harrold (1999).** “Test Case Prioritization: An Empirical Study.” In: *Proceedings of the 15th International Conference on Software Maintenance (ICSM’99)*. IEEE (cited on p. 57, 58, 162).
- Rothermel, G., R. H. Untch, C. Chu, M. J. Harrold (2001).** “Prioritizing Test Cases for Regression Testing.” In: *Transactions on Software Engineering (TSE)* 27.10. IEEE (cited on p. 57, 58, 163).
- Rott, J., R. Niedermayr, E. Juergens, D. Pagano (2017).** “Ticket Coverage: Putting Test Coverage into Context.” In: *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics (WETSoM’17)*. IEEE (cited on p. 29, 157).
- Runeson, P. (2006).** “A Survey of Unit Testing Practices.” In: *Software* 23.4. IEEE (cited on p. 24).
- Scanniello, G., C. Gravino, A. Marcus, T. Menzies (2013).** “Class Level Fault Prediction Using Software Clustering.” In: *Proceedings of the 28th International Conference on Automated Software Engineering (ASE’13)*. IEEE (cited on p. 55).



- Schuler, D., A. Zeller (2009).** “Javalanche: Efficient Mutation Testing for Java.” In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*. ACM (cited on p. 66).
- (2011). “Assessing Oracle Quality with Checked Coverage.” In: *Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST'11)* (cited on p. 14, 45, 46).
  - (2013a). “Checked Coverage: An Indicator for Oracle Quality.” In: *Software Testing, Verification and Reliability (STVR) 23.7*. Wiley Online Library (cited on p. 45, 46).
  - (2013b). “Covering and Uncovering Equivalent Mutants.” In: *Software Testing, Verification and Reliability (STVR) 23.5*. Wiley Online Library (cited on p. 33, 63, 65, 66).
- Schuler, D., V. Dallmeier, A. Zeller (2009).** “Efficient Mutation Testing by Checking Invariant Violations.” In: *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*. ACM (cited on p. 49).
- Schwartz, A., D. Puckett, Y. Meng, G. Gay (2018).** “Investigating Faults Missed by Test Suites Achieving High Code Coverage.” In: *Journal of Systems and Software (JSS)* 144. Elsevier (cited on p. 42).
- Seemann, M. (2015).** *Code coverage is a useless target measure*. URL: <https://blog.ploeh.dk/2015/11/16/code-coverage-is-a-useless-target-measure> (visited on June 1, 2017) (cited on p. 44).
- Shepperd, M. (1988).** “A Critique of Cyclomatic Complexity as a Software Metric.” In: *Software Engineering Journal* 3.2. IET (cited on p. 130).
- Shin, D., S. Yoo, M. Papadakis, D.-H. Bae (2019).** “Empirical evaluation of mutation-based test case prioritization techniques.” In: *Software Testing, Verification and Reliability (STVR) 29.1–2*. Wiley Online Library (cited on p. 58).
- Shippy, T., T. Hall, S. Counsell, D. Bowes (2016).** “So You Need More Method Level Datasets for Your Software Defect Prediction?: Voilà!” In: *Proceedings of the 10th International Symposium on Empirical Software Engineering and Measurement (ESEM'16)*. ACM (cited on p. 55, 128, 135, 138).

- Shivaji, S., E. J. Whitehead, R. Akella, S. Kim (2013).** “Reducing Features to Improve Code Change-Based Bug Prediction.” In: *Transactions on Software Engineering (TSE)* 39.4. IEEE (cited on p. 134).
- Shneiderman, B. (1992).** “Tree Visualization with Tree-Maps: 2-D Space-Filling Approach.” In: *Transactions on Graphics (TOG)* 11.1. ACM (cited on p. 81).
- Silva, D., R. Rabelo, M. Campanha, P. S. Neto, P. A. Oliveira, R. Britto (2016).** “A Hybrid Approach for Test Case Prioritization and Selection.” In: *Proceedings of the Congress on Evolutionary Computation (CEC’16)*. IEEE (cited on p. 57).
- Simon, G. J., V. Kumar, P. W. Li (2011).** “A Simple Statistical Model and Association Rule Filtering for Classification.” In: *Proceedings of the 17th International Conference on Knowledge Discovery and Data Mining (SIGKDD’11)*. ACM (cited on p. 39).
- Singh, Y., A. Kaur, B. Suri (2010).** “Test Case Prioritization Using Ant Colony Optimization.” In: *SIGSOFT Software Engineering Notes (SEN)* 35.4. ACM (cited on p. 57).
- Skinner, J. (2010).** *The Myth of Code Coverage*. URL: <https://jeremyskinner.co.uk/2010/12/10/the-myth-of-code-coverage> (visited on June 1, 2017) (cited on p. 44).
- Smith, B., L. A. Williams (2008).** *A Survey on Code Coverage as a Stopping Criterion for Unit Testing*. Tech. rep. North Carolina State University, Department of Computer Science (cited on p. 64).
- Song, Q., M. Shepperd, M. Cartwright, C. Mair (2006).** “Software Defect Association Mining and Defect Correction Effort Prediction.” In: *Transactions on Software Engineering (TSE)* 32.2. IEEE (cited on p. 40).
- Spieker, H., A. Gotlieb, D. Marijan, M. Mossige (2017).** “Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration.” In: *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA’17)*. ACM (cited on p. 58).
- Shrivastava, A., J. Thiagarajan (2002).** “Effectively Prioritizing Tests in Development Environment.” In: *SIGSOFT Software Engineering Notes (SEN)* 27.4. ACM (cited on p. 166).

- Staats, M., G. Gay, M. Whalen, M. Heimdahl (2012)**. “On the Danger of Coverage Directed Test Case Generation.” In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE’12)*. Springer (cited on p. 44).
- StackOverflow (2019)**. *Developer Survey Results 2019*. URL: <https://insights.stackoverflow.com/survey/2019> (visited on July 5, 2019) (cited on p. 17).
- Strug, J., B. Strug (2012)**. “Machine Learning Approach in Mutation Testing.” In: *Proceedings of the 24th International Conference on Testing Software and Systems (ICTSS’12)*. Springer (cited on p. 50).
- **(2018)**. “Evaluation of the Prediction-Based Approach to Cost Reduction in Mutation Testing.” In: *Proceedings of the 39th International Conference on Information Systems Architecture and Technology (ISAT’18)*. Springer (cited on p. 50).
- Suri, B., S. Singhal (2011)**. “Implementing Ant Colony Optimization for Test Case Selection and Prioritization.” In: *International Journal on Computer Science and Engineering* 3.5. Engg Journals Publications (cited on p. 57).
- Taipale, O., J. Kasurinen, K. Karhu, K. Smolander (2011)**. “Trade-Off Between Automated and Manual Software Testing.” In: *International Journal of System Assurance Engineering and Management (SREQOM)* 2.2. Springer (cited on p. 22).
- Tat, M. J. (2017)**. *Seeing the Random Forest from the Decision Trees: An Explanation of Random Forest*. URL: <https://towardsdatascience.com/seeing-the-random-forest-from-the-decision-trees-an-intuitive-explanation-of-random-forest-beaa2d6a0d80> (visited on May 6, 2019) (cited on p. 38).
- Tengeri, D., L. Vidács, Á. Beszédes, J. Jász, G. Balogh, B. Vancsics, T. Gyimóthy (2016)**. “Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density.” In: *Proceedings of the 9th International Conference on Software Testing, Verification and Validation Workshops (ICSTW’16)*. IEEE (cited on p. 44).
- Tikir, M. M., J. K. Hollingsworth (2002)**. “Efficient Instrumentation for Code Coverage Testing.” In: *SIGSOFT Software Engineering Notes (SEN)*. Vol. 27. 4. ACM (cited on p. 29).
- Tillmann, N., W. Schulte (2005)**. “Parameterized Unit Tests.” In: *SIGSOFT Software Engineering Notes (SEN)* 30.5. ACM (cited on p. 23).

- Torgo, L (2010).** *Data Mining with R, learning with case studies*. Chapman and Hall / CRC (cited on p. 136, 142).
- Turhan, B., T. Menzies, A. B. Bener, J. Di Stefano (2009).** “On the Relative Value of Cross-Company and Within-Company Data for Defect Prediction.” In: *Empirical Software Engineering* 14.5. Springer (cited on p. 56, 128, 137).
- Usaola, M. P., P. R. Mateo (2010).** “Mutation Testing Cost Reduction Techniques: A Survey.” In: *Software* 27.3. IEEE (cited on p. 30).
- Vera-Pérez, O. L., B. Danglot, M. Monperrus, B. Baudry (2017).** “A Comprehensive Study of Pseudo-Tested Methods.” In: *Empirical Software Engineering*. Springer (cited on p. 53, 54, 66, 92, 97, 98, 174).
- Vera-Pérez, O. L., M. Monperrus, B. Baudry (2018).** “Descartes: A PITest Engine to Detect Pseudo-Tested Methods.” In: *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE’18)*. ACM (cited on p. 74, 76, 96, 112).
- Voas, J. M., G. McGraw (1997).** *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons (cited on p. 51).
- Vocke, H. (2018).** *The Practical Test Pyramid*. URL: <https://martinfowler.com/articles/practical-test-pyramid.html> (visited on May 17, 2019) (cited on p. 25, 163).
- Wacker, M. (2015).** *Just Say No to More End-to-End Tests*. URL: <https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html> (visited on May 2, 2019) (cited on p. 26).
- Wah, K. H. T. (2000).** “A Theoretical Study of Fault Coupling.” In: *Software Testing, Verification and Reliability (STVR)* 10.1. Wiley Online Library (cited on p. 31).
- (2003). “An Analysis of the Coupling Effect I: Single Test Data.” In: *Science of Computer Programming* 48.2-3. Elsevier (cited on p. 31).
- Walcott, K. R., M. L. Soffa, G. M. Kapfhammer, R. S. Roos (2006).** “Time-Aware Test Suite Prioritization.” In: *Proceedings of the 15th International Symposium on Software Testing and Analysis (ISSTA’06)*. ACM (cited on p. 57).

- Wei, Y., B. Meyer, M. Oriol (2010).** “Is Branch Coverage a Good Measure of Testing Effectiveness?” In: *Empirical Software Engineering and Verification*. Springer (cited on p. 43).
- Weyuker, E. J., T.J. Ostrand (2008).** “What Can Fault Prediction Do for YOU?” In: *Proceedings of the 2nd International Conference on Tests and Proofs (TAP’08)*. Springer (cited on p. 55, 128).
- Witten, I. H., E. Frank, M. A. Hall, C. J. Pal (2016).** *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers (cited on p. 142).
- Wong, W. E., J. R. Horgan, S. London, A. P. Mathur (1994).** “Effect of Test Set Size and Block Coverage on the Fault Detection Effectiveness.” In: *Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE’94)* (cited on p. 42).
- Wong, W. E., J. R. Horgan, S. London, H. Agrawal (1997).** “A Study of Effective Regression Testing in Practice.” In: *Proceedings of the 8th International Symposium On Software Reliability Engineering (ISSRE’97)*. IEEE (cited on p. 57, 163).
- Xia, X., D. Lo, S. J. Pan, N. Nagappan, X. Wang (2016).** “HYDRA: Massively Compositional Model for Cross-Project Defect Prediction.” In: *Transactions on Software Engineering (TSE)* 42.10. IEEE (cited on p. 55, 56, 128).
- Xls, K. (2017).** *False test coverage*. URL: <https://ondergetekende.nl/false-test-coverage.html> (visited on June 1, 2017) (cited on p. 44).
- Xu, Z., J. Liu, X. Luo, T. Zhang (2018).** “Cross-Version Defect Prediction Via Hybrid Active Learning With Kernel Principal Component Analysis.” In: *Proceedings of the 25th International Conference on Software Analysis, Evolution, and Reengineering (SANER’18)*. IEEE (cited on p. 56, 128).
- Yang, Q., J. J. Li, D. M. Weiss (2009).** “A Survey of Coverage-Based Testing Tools.” In: *The Computer Journal* 52.5. Oxford University Press (cited on p. 14, 64).
- Yoo, S., M. Harman (2012).** “Regression Testing Minimization, Selection and Prioritization: A Survey.” In: *Software Testing, Verification and Reliability (STVR)* 22.2. Wiley Online Library (cited on p. 27, 56, 163).

- Zafar, H., Z. Rana, S. Shamail, M. M. Awais (2012).** “Finding Focused Itemsets from Software Defect Data.” In: *Proceedings of the 15th International Multitopic Conference (INMIC'12)*. IEEE (cited on p. 40).
- Zelkowitz, M. V. (2003).** *Advances in Computers: Highly Dependable Software*. Academic Press (cited on p. 22).
- Zhang, F., Q. Zheng, Y. Zou, A. E. Hassan (2016).** “Cross-Project Defect Prediction Using a Connectivity-Based Unsupervised Classifier.” In: *Proceedings of the 38th International Conference on Software Engineering (ICSE'18)*. ACM (cited on p. 56, 128).
- Zhang, H., X. Zhang, M. Gu (2007).** “Predicting Defective Software Components From Code Complexity Measures.” In: *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC'07)*. IEEE (cited on p. 128).
- Zhang, J., L. Zhang, M. Harman, D. Hao, Y. Jia, L. Zhang (2018).** “Predictive Mutation Testing.” In: *Transactions on Software Engineering (TSE)*. IEEE (cited on p. 50, 111, 113, 121).
- Zhang, L. (2018).** “Hybrid Regression Test Selection.” In: *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. IEEE (cited on p. 162).
- Zhang, L., D. Marinov, L. Zhang, S. Khurshid (2012).** “Regression Mutation Testing.” In: *Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA'12)*. ACM (cited on p. 49).
- Zhang, Y., A. Mesbah (2015).** “Assertions are Strongly Correlated with Test Suite Effectiveness.” In: *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (cited on p. 47).
- Zhi, J., V. Garousi (2013).** “On Adequacy of Assertions in Automated Test Suites: An Empirical Investigation.” In: *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13)*. IEEE (cited on p. 22).
- Zhu, H., P. A. Hall, J. H. May (1997).** “Software Unit Test Coverage and Adequacy.” In: *Computing Surveys (CSUR)* 29.4. ACM (cited on p. 14, 29–31).

- Zhu, Q., A. Panichella, A. Zaidman (2018).** “A Systematic Literature Review of How Mutation Testing Supports Quality Assurance Processes.” In: *Software Testing, Verification and Reliability (STVR)* 28.6. Wiley Online Library (cited on p. 32).
- Zimmermann, J. (2017).** *Why You Need Test Coverage, But Shouldn't Trust It*. URL: <http://www.developintelligence.com/blog/2017/11/why-test-coverage-shouldnt-trust> (visited on June 1, 2017) (cited on p. 44).
- Zimmermann, T., N. Nagappan (2008).** “Predicting Defects using Network Analysis on Dependency Graphs.” In: *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. IEEE (cited on p. 55).
- Zimmermann, T., R. Premraj, A. Zeller (2007).** “Predicting Defects for Eclipse.” In: *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering (PROMISE'07)*. IEEE Computer Society, p. 9 (cited on p. 55, 132, 135).
- Zimmermann, T., N. Nagappan, H. Gall, E. Giger, B. Murphy (2009).** “Cross-Project Defect prediction: A Large Scale Experiment on Data vs. Domain vs. Process.” In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*. ACM (cited on p. 37, 55, 56, 128, 137, 159).
- Zörner, S. (2012).** *Softwarearchitekturen dokumentieren und kommunizieren*. Hanser (cited on p. 81).
- ndepend (2017).** *Code Metrics Definitions*. URL: <https://www.ndepend.com/docs/code-metrics#ILNestingDepth> (visited on Apr. 4, 2019) (cited on p. 132).





# LIST OF FIGURES

2.1 Overview of the testing terminology of automated tests. . . . .	23
2.2 The test pyramid. . . . .	26
2.3 Relation between fault, defect, and failure. . . . .	28
2.4 10-fold cross-validation. . . . .	37
2.5 Cross-project prediction validation. . . . .	37
4.1 Overview of the mutation process. . . . .	72
5.1 Boxplots comparing the proportion of pseudo-tested methods between projects with unit tests and projects with integration tests. . . . .	82
5.2 Distribution of pseudo-tested methods in TWITTER GRAPHJET. . . . .	82
5.3 Distribution of pseudo-tested methods in JABREF. . . . .	83
5.4 Classification of pseudo-tested methods by their functional category. . . . .	91
5.5 Proportion of methods that are pseudo-tested out of the mu- tated methods per return type. . . . .	94
6.1 Example illustrating the minimal stack distance. . . . .	103
6.2 Overview of the stack distance computation. . . . .	104

6.3	Proportion of pseudo-tested methods and of executed mutants per minimal stack distance value. . . . .	117
6.4	Project characteristics affecting the correlation strength between a method's minimal stack distance and its mutation testing verdict. . . . .	118
6.5	Exemplary variable importances of the models to predict pseudo-tested methods. . . . .	121
6.6	Duration of analyses (in hours) and slowdown factor based on the normal test suite execution. . . . .	124
7.1	Overview of the approach to identify low-fault-risk methods. . . . .	132
7.2	Derivation of faulty methods. . . . .	140
7.3	Example of a method change without behavior modification to address API changes. . . . .	141
7.4	Metrics M1 to M5 are not normally distributed. . . . .	141
7.5	Influence of the number of selected rules. . . . .	144
7.6	Comparison of within- and cross-project IDP classifiers. . . . .	151
7.7	Comparison of the classification classes of defect prediction and IDP. . . . .	158
8.1	Overview of the steps of the Test Impact Analysis (TIA). . . . .	165
8.2	Class diagram illustrating how pseudo-testedness information influences the score that a test case achieves by covering a method. . . . .	168

# LIST OF TABLES

4.1 Duration of a mutation analysis using PIT (in hours) compared to a regular test suite execution. . . . .	63
4.2 Return values for primitive types and string. . . . .	68
5.1 Study objects. . . . .	79
5.2 Pseudo-tested methods in the study objects. . . . .	81
5.3 Functional method categories. . . . .	89
5.4 Results of the correlation tests. . . . .	95
6.1 Study objects. . . . .	109
6.2 Example of a full mutation matrix. . . . .	112
6.3 Return values of the Descartes implementation of the operator. . . . .	113
6.4 Overview of the mutation analysis results. . . . .	114
6.5 Correlation coefficient between a method's minimal stack distance and its mutation testing verdict. . . . .	115
6.6 Performance when predicting a method's mutation testing verdict. . . . .	119
6.7 Performance when predicting pseudo-tested methods. . . . .	120

6.8 Performance when predicting the mutation testing verdict of a test-case method pair. . . . .	122
7.1 Computed metrics for each method. . . . .	131
7.2 Study objects from the Defects4J data set. . . . .	139
7.3 Generated classes and their value ranges. . . . .	142
7.4 Code and change metrics used in Giger <i>et al.</i> (2012). . . . .	146
7.5 Evaluation of within-project IDP to identify low-fault-risk methods. . . . .	147
7.6 Top five association rules for LANG. . . . .	148
7.7 Evaluation of cross-project IDP. . . . .	150
7.8 Top five association rules computed on all projects. . . . .	151
7.9 Results of a traditional defect prediction approach. . . . .	153
8.1 Comparison of different approaches in respect to the number of test cases and their duration until the first test failure. . . . .	170

# LIST OF LISTINGS

2.1 Example of an equivalent mutant. . . . .	32
3.1 Test case with a not very effective assertion. . . . .	47
4.1 Example of a mutant that returns a non-primitive value. . . . .	69
5.1 Example of a pseudo-tested method in <code>JFREECHART</code> . . . . .	84
5.2 Test case covering the pseudo-tested method from Listing 5.1.	84
5.3 Example of a pseudo-tested method in <code>CONQAT LIB COMMONS</code> .	85
5.4 Test case covering the pseudo-tested method from Listing 5.3.	86
5.5 Example of a pseudo-tested method in <code>APACHE COMMONS MATH</code> .	86
5.6 Test case covering the pseudo-tested method from Listing 5.5.	87
6.1 Example of an instrumented method. . . . .	105



# LIST OF ABBREVIATIONS

<b>AST</b>	abstract syntax tree
<b>CI</b>	continuous integration
<b>HIL</b>	hardware-in-the-loop
<b>I/O</b>	input/output
<b>IDE</b>	integrated development environment
<b>IDP</b>	inverse defect prediction
<b>JDK</b>	Java Development Kit
<b>JVM</b>	Java Virtual Machine
<b>LFR</b>	low fault risk
<b>LOC</b>	lines of code
<b>PIT</b>	Pitest
<b>QA</b>	quality assurance

<b>SLOC</b>	source lines of code
<b>SMOTE</b>	synthetic minority over-sampling technique
<b>TCE</b>	Trivial Compiler Equivalence
<b>TFF</b>	time to first failure
<b>TGA</b>	Test Gap Analysis
<b>TIA</b>	Test Impact Analysis
<b>UI</b>	user interface
<b>VCS</b>	version control system
<b>XML</b>	extensible markup language



