An Analysis of the Differences between Unit and Integration Tests

Dissertation
zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades
"Doctor rerum naturalium"
der Georg-August-Universität Göttingen

im Promotionsprogramm Computer Science (PCS)
der Georg-August University School of Science (GAUSS)

vorgelegt von

Fabian Trautsch
aus Göttingen

Göttingen, März 2019

Betreuungsausschuss

Prof. Dr. Jens Grabowski,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Marcus Baum,
Institut für Informatik, Georg-August-Universität Göttingen

Dr. Steffen Herbold,
Institut für Informatik, Georg-August-Universität Göttingen

Mitglieder der Prüfungskommission

Referent:	Prof. Dr. Jens Grabowski, Institut für Informatik, Georg-August-Universität Göttingen
Korreferent:	Prof. Dr. Marcus Baum, Institut für Informatik, Georg-August-Universität Göttingen
Korreferent:	Prof. Dr.-Ing. Ina Schieferdecker, Institut für Telekommunikationssysteme, Technische Universität Berlin

Weitere Mitglieder der Prüfungskommission

Dr. Steffen Herbold,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr.-Ing. Delphine Reinhardt
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Carsten Damm
Institut für Informatik, Georg-August-Universität Göttingen

Tag der mündlichen Prüfung
08.04.2019

Abstract

Context: In software testing, there are several concepts that were established over the years, including unit and integration testing. These concepts are defined in standards and used in software testing certifications, which underline their importance for research and industry. However, these concepts are decades old. Nowadays, we do not have any evidence that these concepts still apply for modern software systems.

Objective: The purpose of this thesis is to evaluate, if the differences between unit and integration testing are still valid nowadays. To this aim, we analyze defined differences between these test levels to provide evidence, if these are still current in modern software.

Method: We performed quantitative and qualitative analysis on differences between unit and integration tests. The quantitative analysis was performed via a case study including 27 Java and Python projects with more than 49000 tests. During this analysis we classified tests into unit and integration tests according to the definitions of the Institute of Electrical and Electronics Engineers (IEEE) and International Software Testing Qualification Board (ISTQB) and calculated several metrics for those tests. We then used these metrics to assess three differences between these levels. For the qualitative analysis we searched for relevant research literature, developer comments, and further information regarding differences between unit and integration tests. The found resources are evaluated to gain an understanding of the research and industrial perspective on the differences, i.e., if they are existent and to which magnitude.

Results: We found that more integration than unit tests are present in most projects, when classified according to the definitions of the IEEE and ISTQB. However, the exact numbers differ between these definitions. Based on the developer classification of tests, there is no significant difference in the number of unit and integration tests. Our quantitative analysis highlights that diverse defined differences are no longer existent. We found, that the defect types that are detected by both test types, do not differ from each other and that there are no significant differences in their execution time. However, we confirmed that unit tests are better able to pinpoint the source of a defect. Our qualitative analysis of research and industrial perspective shows, that both test types are executed automatically, that their test objectives mostly differ from each other, and that practitioners experienced that integration tests are more costly than unit tests.

Conclusions: Our results suggest that the current definitions of unit and integration tests are outdated and need to be reconsidered as most of the differences are vanishing. One reason for this could be technological advancements in the area of software testing and software engineering. However, this needs to be further investigated.

Zusammenfassung

Kontext: Im Gebiet des Softwaretestens wurden über die Jahre verschiedene Konzepte, wie Unit- und Integrationstests, etabliert. Diese Konzepte wurden in Standards definiert und werden auch heutzutage noch in Softwaretesten Zertifikaten benutzt. Dies unterstreicht ihre Wichtigkeit für die Industrie und Forschung. Allerdings sind diese Konzepte schon Jahrzehnte alt. Aktuell existiert keine Evidenz, ob diese Konzepte noch immer für moderne Software Systeme zutreffen.

Ziel: Das Ziel dieser Arbeit ist die Evaluation, ob die Unterschiede zwischen Unit- und Integrationstests, wie sie in der Standardliteratur beschrieben werden, noch immer zutreffen. Dazu analysieren wir die Unterschiede zwischen diesen beiden Testarten.

Methode: Wir benutzen qualitative und quantitative Methoden in dieser Arbeit. Die quantitative Analyse umfasst die Ausführung einer Fallstudie mit 27 Java und Python Projekten, welche insgesamt mehr als 49000 Tests beinhalten. Innerhalb dieser Analyse klassifizieren wir alle Tests in Unit- bzw. Integrationstests mittels der Definitionen der Institute of Electrical and Electronics Engineers (IEEE) und International Software Testing Qualification Board (ISTQB). Zudem berechnen wir mehrere Metriken für diese Tests, um die Unterschiede zu quantifizieren. Für die qualitative Analyse haben wir relevante Literatur, Entwicklerkommentare, und weitere Informationen die sich mit den Unterschieden zwischen Unit- und Integrationstests befassen, analysiert.

Ergebnisse: Unsere Ergebnisse zeigen, dass mehr Integrations- als Unittests in aktuellen Projekten vorhanden sind, wenn wir die Tests nach den Definitionen des IEEE und des ISTQB klassifizieren. Die exakte Anzahl hängt von der Definition ab. Wenn wir die Tests so klassifizieren wie ihre Entwickler, sind nicht mehr Integrations- als Unitttests vorhanden. Die quantitative Analyse hat gezeigt, dass die meisten in der Literatur genannten Unterschiede zwischen beiden Testarten für moderne Software nicht mehr zutreffen. Unsere Ergebnisse zeigen, dass Unit- und Integrationstests dieselben Arten von Fehlern entdecken und dass es keine Unterschiede in ihrer Ausführungszeit gibt. Allerdings konnten wir bestätigen, dass Unittests besser zur Lokalisierung von Fehlern geeignet sind. Unsere qualitative Analyse hat gezeigt, dass beide Testarten automatisch ausgeführt werden, ihr Testziel sich voneinander unterscheidet und das Entwickler Integrationstests als teurer wahrnehmen.

Schlussfolgerung: Unsere Ergebnisse zeigen, dass viele Unterschiede zwischen Unit- und Integrationstests nicht mehr vorhanden sind. Dies suggeriert, dass die derzeit geltenden Definitionen von Unit- und Integrationstests nicht für moderne Software Systeme zutreffen. Ein Grund hierfür könnte die Evolution der Softwareentwicklung sein, welche durch die Verbesserung und Entwicklung von Softwaretesten-Werkzeugen vorangetrieben wird.

Acknowledgements

There are several people that I would like to thank, as they have supported me throughout my whole thesis. First of all, I want to thank my first supervisor Prof. Dr. Jens Grabowski, which made a lot of things possible during my time in his research group. Not only the possibility to pursue my PhD in the area that I like, but also other things like the traveling to conferences, research stays in Beijing and Shanghai, and the possibility to participate in a Summer School. Furthermore, he always gave me feedback, discussed the work, and gave me guidance throughout my PhD. In addition, I want to thank my second supervisor Prof. Dr. Marcus Baum. He gave me valuable feedback and asked the right questions during my presentations to lead me to my goal. Furthermore, I would also like to thank Prof. Dr.-Ing. Ina Schieferdecker, Prof. Dr.-Ing. Delphine Reinhardt, and Prof. Dr. Carsten Damm for investing their time for me.

Another special thanks goes out to Steffen Herbold. He did not only supervised my PhD thesis, but also my Master Thesis and provided me with a lot of good feedback, discussions, and helpful comments on all the work that I have done. He did not only made it possible that I got the MINT award for the best Master's Thesis, but also supported me during my whole PhD time.

Additionally, I dedicate many thanks to all my current and former colleagues in my research group and the institute. They helped me with a lot of good discussions and the proof-reading of all the things that I have written. During my time in this research group I did not only find colleagues, but a lot of new friends which I really enjoyed spending my time with. Another special thanks goes to Patrick Harms, who basically made my PhD possible by supervising my Bachelor Thesis... and let it pass. Furthermore, I would specially thank my brother Alexander Trautsch, Johannes Erbel, Patrick Harms, Ella Albrecht, and Philipp Makedonski for reviewing and commenting on this thesis. While it is sometimes hard to rewrite a certain paragraph for the 10th time, it is (often) the right thing to do to improve it.

Furthermore, I would also want to thank my aunt and uncle that have always supported me during my steps in my education. Furthermore, I want to thank my brother, who made it possible for me to do some internships in his work place, which then encouraged me to follow the path of a computer scientist. Finally, I want to thank my wife Irina Trautsch. Not only for the discussion that we had about my thesis, but also the support that she gave me all the long way from my Masters Thesis till my PhD. I do not want to miss her in my life. I want to dedicate my thesis to my parents, which died way too early.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

API Application Programming Interface.

AST Abstract Syntax Tree.

CAS Changes on Assignment Statements.

CBS Changes on Branch Statements.

CC Cyclomatic Complexity.

CDDI Data Declaration and Definition.

CE Changed Entity.

CFC Changes on Function Call.

CFDD Changes on Function Declaration/Definition.

CI Continuous Integration.

CLS Changes on Loop Statements.

CO Others.

COD Changes on Documentation.

COMFORT COllection of Metrics FOR Tests.

CPA Clean Program Assumption.

CPD Changes on Preprocessor Directives.

CRGS Changes on Return/Goto Statements.

CSV Comma Separated Values.

CT Change Type.

DCD Defect Call Depth.

FWER Family-wise Error Rate.

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen.

HPC High Performance Computing.

ICST International Conference on Software Testing, Verification, and Validation.

IDE Integrated Development Environment.

IEEE Institute of Electrical and Electronics Engineers.

ISSTA International Symposium on Software Testing and Analysis.

ISTQB International Software Testing Qualification Board.

ITS Issue Tracking System.

JVM Java Virtual Machine.

KLOC Thousand Lines of Code.

LLOC Logical Lines of Code.

LOC Lines of Code.

MC/DC Modified Condition/Decision Coverage.

MI Maintainability Index.

ms Milliseconds.

ODC Orthogonal Defect Classification.

OO Object-Oriented.

PE Parent Entity.

pKLOC Thousand Production Lines of Code.

pLOC Production Lines of Code.

RQ Research Question.

SLR Systematic Literature Review.

SUT System Under Test.

TDD Test-Driven Development.

TestKLOC Thousand Test Lines of Code.

TestLOC Test Lines of Code.

UML Unified Modeling Language.

VCS Version Control System.

VR Virtual Reality.

1. Introduction

Nowadays, software testing gets more and more important through the huge number of different software systems and their direct impact on our lives [1]. The research area of software testing is growing due to the increasing complexity of software systems, including the development and advancement of dedicated software testing conferences like the International Conference on Software Testing, Verification, and Validation (ICST) [2] or the International Symposium on Software Testing and Analysis (ISSTA) [3]. Most recently, the area of evidence-based software testing [4, 5] gets attention, empowered through the availability of data from open-source software projects [6, 7]. One challenge of evidence-based software testing is the creation of a body of evidence for important aspects in software testing [8]. Unfortunately, most of the techniques and theories that are present in the standard literature [9, 1, 10, 11, 12, 13, 14, 15] and taught at universities rely on anecdotes, are inconsistent [16], or not based on empirical evidence [8].

Within this thesis we provide evidence for a topic, which is highly important in research and practice: the separation of tests on different test levels and their differences. That test levels are an important concept is highlighted by the large number of publications that are focused on these topics, as well as their incorporation within several development models like, e.g., the V-model [9] or the Waterfall model [17]. Hereby, unit testing is one of the most advanced areas within the research on software test levels. There exist work on the generation of tests [18, 19, 20], minimization of test cases [21], or the detection of test refactorings on the unit level [22, 23]. However, some research is done in the field of integration testing, too. For example, on the automation of integration testing [24], test case prioritization [25], or how to profit from unit tests for integration testing [26]. The definitions of unit and integration tests, e.g., by the Institute of Electrical and Electronics Engineers (IEEE) [27] or International Software Testing Qualification Board (ISTQB) [28], are used in software testing certifications.

Another aspect that highlights the high interest in this topic, is a current proposal that is made within the development community [29]. Instead of testing all parts of a software system on unit level with some integration tests, as described in the software testing standard literature [9, 1, 10, 11], the proposal states that only some unit tests should be created (for the most difficult parts) while most software parts should be tested via integration tests instead. The reasons for this shift of the software testing paradigm are manifold. Developers state that the creation of unit tests is not really worth the effort compared to their effectiveness, as mocks need to be created and/or the design of the software needs to be adapted so that unit tests are applicable [30, 31]. Furthermore, developers argue that integration tests are

more realistic than unit tests, as they test scenarios instead of units, and therefore provide more confidence in the software system [32, 33, 29]. However, the problem that currently exists is best summarized by one of the developers that is engaged in the discussion: "We don't have empirical evidence showing that this is actually true, unfortunately." [32]. Hence, evidence is missing that could help us to assess if this kind of proposal is problematic or if it improves the software testing process. Therefore, within this thesis, we focus on the differences between unit and integration tests and bridge the mentioned gap by providing evidence to assess if the differences are still valid in modern software development contexts. This could also give us hints if the use of the decades old definitions of unit and integration tests still fit to separate between those test types.

1.1. Scope of the Thesis

In this thesis, we present an study on unit and integration tests that illuminates two aspects: the distribution of unit and integration tests in open-source projects and the differences between these test levels. Furthermore, to widen our scope and improve the external validity of our results, we perform our study based on two different definitions of unit and integration tests: the definitions of the IEEE and ISTQB.

To steer our research, we investigate and answer several Research Questions (RQs) for each of these aspects in this thesis. The differences between unit and integration tests are analyzed quantitatively and qualitatively using data collected from the repositories of several open-source projects, as well as publications and other textual resources.

Distribution of Unit and Integration Tests
At first, we need to be able to detect unit and integration tests and assess if and how they are developed and used. Furthermore, we provide empirical evidence on the distribution of unit and integration tests to evaluate if the proposed software testing paradigm shift explained above (i.e., more integration than unit tests) is already visible in practice. Hence, we define the following RQ:

- **RQ 1:** What is the distribution of unit and integration tests in open-source projects?

This question leads to the following more detailed sub questions, focusing on different aspects of the distribution of unit and integration tests in open-source projects:

- **RQ 1.1:** To what extend is the trend of developing more integration than unit tests visible in open-source projects?
- **RQ 1.2:** How are unit and integration tests distributed, if we reuse the developer classification of tests?
- **RQ 1.3:** To what extend are developers classifying unit and integration tests according to the definitions?

Identifier	Difference	Source	Analysis Type
D1	Lower execution time of unit tests	[10]	Quantitative
D2	Unit tests detect different defects than integration tests	[9, 1, 11, 12, 13]	Quantitative
D3	Unit tests directly pinpoint the source of the problem	[9, 10]	Quantitative
D4	The execution of unit tests is easily automatable	[11, 14, 15]	Qualitative
D5	Unit and integration tests have different test objectives	[9, 1, 10, 11, 12, 13]	Qualitative
D6	Unit tests cost less than integration tests	[13]	Qualitative

Table 1.1.: Differences between unit and integration tests as stated in the standard literature, together with their source.

In RQ 1.1 we evaluate whether there are more integration than unit tests in open-source projects using the definitions of the IEEE and ISTQB, while in RQ 1.2 we assess the number of unit and integration tests according to the developers classification. Then, in RQ 1.3, we compare the results of RQ 1.1 and RQ 1.2 with each other. We want to assess, if developers are classifying unit and integration tests according to the definitions.

Differences between Unit and Integration Tests
In the second part of our research, we evaluate the differences between unit and integration tests. Therefore, we investigate the following RQ:

- **RQ 2:** What are the differences between unit and integration tests?

To compile a list of differences between unit and integration tests, we assessed software engineering and software testing text books used for teaching and education [9, 1, 10, 11, 12, 13, 14, 15].

Table 1.1 highlights our collected differences, together with their source. Overall, we identified six differences from the standard literature. However, not all of them can be analyzed quantitatively due to missing data, e.g., there is no cost data for open-source projects available that track the money spend for the design, development, and execution of tests. Therefore, we use a mixture of quantitative and qualitative analysis techniques to assess these differences. As noted in Table 1.1, differences D1-D3 are analyzed quantitatively. Hereby, we assess these differences by collecting and mining data from open-source projects and analyze this data statistically to assess the differences. D4-D6 are evaluated

based on a qualitative analysis. Within this analysis, we searched for relevant research literature, developer comments, and further internet resources regarding these differences. The found resources are evaluated to gain an understanding of the research and industrial perspective on the differences, i.e., if these differences are existent and to which extent. We do not only focus on research literature, but include developer comments and the current industrial landscape, e.g., companies that provide services, current frameworks, or libraries, to analyze these differences. The difference regarding the test objective (D5) is rather clear, as unit and integration tests have different objectives by definition. However, some literature (e.g., [9]) state, that unit tests are used to test the robustness, efficiency, and maintainability of a software (in contrast to an integration test).

Based on the differences between unit and integration tests, highlighted in Table 1.1, we defined several sub-RQs for RQ 2. These RQs are listed below:

- **RQ 2.1:** What are the differences between unit and integration tests in open-source projects in terms of their execution time? (D1)
- **RQ 2.2:** What are the differences between unit and integration tests in open-source projects in terms of their effectiveness? (D2)
- **RQ 2.3:** What are the differences between unit and integration tests in open-source projects in terms of their defect-locality? (D3)
- **RQ 2.4:** What are the differences between unit and integration tests in terms of their execution automation? (D4)
- **RQ 2.5:** What are the differences between unit and integration tests in terms of their test objective? (D5)
- **RQ 2.6:** What are the differences between unit and integration tests in terms of their costs? (D6)

RQs 2.1-2.3 are focused on open-source projects, as we perform our quantitative analysis using data mined from those projects. Unfortunately, industrial data was not available. RQs 2.4-2.6 are analyzed qualitatively. Hence, we are not focused on open-source projects alone, but evaluate the current research literature and other internet resources regardless of the studied projects.

1.2. Goals and Contributions

This thesis advances the state of the art and the body of knowledge in the fields of software testing and evidence-based software testing through the following contributions:

- An approach to **classify software tests** into unit and integration tests (Section 4.2.4). This novel approach is using coverage data of tests to classify software test cases into unit and integration tests. Our approach supports the classification based on the definitions of the IEEE and ISTQB, as well as the classification based on naming

conventions, i.e., the developer classification. This classification approach is the cornerstone for our quantitative analysis.

- An approach and implementation to **extract the defect-locality of software tests** (Section 4.2.8). This approach is using artificial defects, integrates them into the program, and assesses the depth of the call stack that was generated till the defect was detected by a test. It makes use of the Java instrumentation Application Programming Interface (API) [34] and is therefore applicable to all Java programs that support this API. This approach is used within our quantitative analysis to compare the defect-locality of unit and integration tests to assess whether a unit test can directly pinpoint the source of a defect.

- A **quantitative analysis of the distribution of unit and integration tests** in open-source projects (Section 5). This is the result of RQ 1. Hence, the analysis comprises of an analysis of the number of unit and integration tests in current open-source projects, an analysis of the developer classification of tests, as well as an analysis of the overlap between the developer classification and classification by definition of unit and integration tests. The results of this analysis are essential for our research and are used for the subsequent RQs.

- A **quantitative analysis** of the **differences** between unit and integration tests, which is done via a case study (Section 6). We collected different data from open-source projects and analyzed them statistically with respect to the **execution time**, **test effectiveness**, and **defect-locality**. Hence, within this analysis we assess the differences between unit and integration tests empirically. The results of this analysis are part of the answer to RQ 2.

- A **qualitative analysis** of the **differences** between unit and integration tests, which is done by assessing relevant literature and textual resources (Section 7). We analyzed them to gather facts regarding the **execution automation**, **test objective**, and **costs** of unit and integration tests. The results of this analysis form the second part of our answer to RQ 2.

- To facilitate further insights and the replication of our study, we provide our **framework for the mining of data from software repositories** and our **framework for the analysis of software tests** (Section 4.2.10), and a **data set** based on the data used within our case study (Section 4.1). This includes the test classification, their execution time, effectiveness and defect-locality, as well as additional metrics like the covered production lines of code or covered test lines of code. The **mining framework** includes several plug-ins for the collection of data from different software repositories. Furthermore, it is a scalable framework that is able to process and store large amounts of different types of data due to its use of big data technologies. This framework is used to collect the necessary meta-data about projects, which are used in our

analysis of the distribution and differences of unit and integration tests. The framework for the **analysis of software tests** is developed as a standalone framework, but it can cooperate with our data mining framework mentioned above. It includes four different steps from loading the data to storing results. The framework is used to collect the necessary metrics (e.g., defect detection capabilities) for our case study. In addition, this framework can enable other researchers to contribute to the body of knowledge of evidence-based software testing.

1.3. Impact

The results of this dissertation and further research that has been performed to enable this work have been published in one scientific journal article and three peer-reviewed international conference proceedings. Furthermore, the author of this thesis has contributed to one book chapter. One of the authors conference publications was awarded with an "ACM SIGSOFT Distinguished Paper Award" (see below).

Journal Articles

- F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski. "Addressing problems with replicability and validity of repository mining studies through a smart data platform" in *Empirical Software Engineering*, vol. 23, no. 2, Springer, 2018. Available: `https://doi.org/10.1007/s10664-017-9537-x`.

  Own contributions
  I am the lead author of this publication. I performed most of the work for this publication including the technical implementations, except the model-based transformation and extraction framework that I reused from Dr. P. Makedonski for the first version of SmartSHARK and the effort prediction implementation from Dr. S. Herbold. Furthermore, I have developed the new version of SmartSHARK and have done the experiments performed with it. The analysis of current problems within case studies with respect to the external validity and the experience reports regarding the feasibility of our developed platform was done together with Dr. S. Herbold.

Conferences

- F. Trautsch. "Reflecting the Adoption of Software Testing Research in Open-Source Projects" in *Proceedings of the 10th International Conference on Software Testing, Verification and Validation (ICST 2017)*, IEEE, 2017, PhD Symposium. Available: `https://doi.org/10.1109/ICST.2017.77`.

  Own contributions
  I am the single author of this publication and performed all work myself.

- F. Trautsch and J. Grabowski. "Are there any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects" in *Proceedings of the 10th International Conference on Software Testing, Verification and Validation (ICST 2017)*, IEEE, 2017. Available: `https://doi.org/10.1109/ICST.2017.26`.

  Own contributions
  I am the lead author of this publication. All main contributions, implementations, and case studies have been done by myself.

- F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski. "Addressing Problems with External Validity of Repository Mining Studies Through a Smart Data Platform" in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR 2016)*, ACM, 2016. Available: `http://doi.acm.org/10.1145/2901739.2901753`. Awarded with an **ACM SIGSOFT Distinguished Paper Award**.

  Own contributions
  I am the lead author of the publication. I performed most of the work for this publication, including the technical implementations, except the model-based transformation and extraction framework that I reused from Dr. P. Makedonski for the described version of SmartSHARK and the effort prediction implementation from Dr. S. Herbold. Furthermore, Dr. S. Herbold contributed to the analysis of current problems within case studies with respect to the external validity and the discussion of these problems.

Book Chapter

- S. Herbold, F. Trautsch, P. Harms, V. Herbold, and J. Grabowski. "Experiences With Replicable Experiments and Replication Kits for Software Engineering Research" in *Advances in Computers*, vol. 113, Elsevier, 2019. Available: `https://doi.org/10.1016/bs.adcom.2018.10.003`.

  Own contributions
  I contributed to this book chapter by an experience report on the analysis of test type characteristics.

Moreover, the author of this dissertation supervised and co-supervised two student projects and two master theses.

Student Projects

- A. Amirfallah. "Literature Survey on Developer Social Networks", Student Project, Institute of Computer Science, University of Goettingen. 2017.

- B. Ledel. "Topic Modeling Literature Survey and Word Clouds", Student Project, Institute of Computer Science, University of Goettingen. 2017.

Master Theses

- A. Khajeh. “Heuristics and machine learning for merging developer identities across multiple software repositories”, Master Thesis, Institute of Computer Science, University of Goettingen. 2018.

- L. Ul Khair. “Change Classification Techniques for Commits using Static Code Analysis and Issue Tracking Data”, Master Thesis, Institute of Computer Science, University of Goettingen. 2018.

1.4. Structure of the Thesis

This thesis covers several aspects related to the RQs stated above. It is structured as follows.

Chapter 2 summarizes the foundations, which are necessary for understanding the rest of this thesis. It includes foundations regarding software testing (Section 2.1), mutation testing (Section 2.2), as well as statistical hypothesis testing (Section 2.3)

Chapter 3 presents related work to the scientific topics to which the author contributed during his studies and puts our work into a broader research context. This chapter includes the related work to the topics of test level classification (Section 3.1), test effectiveness assessment (Section 3.2), defect classification (Section 3.3), defect-locality (Section 3.4), distribution of unit and integration tests (Section 3.5), and works that analyze differences between unit and integration tests (Section 3.6). In addition, a small summary of the related work together with the research delta is given within this chapter (Section 3.7).

Chapter 4 describes our research methodology. Within this chapter, we give a short overview of our methodology (Section 4.1), present the data collection processes for our case study (Section 4.2) and give remarks for the analysis of the data (Section 4.3).

Chapter 5 presents the results of the analysis of the distribution of unit and integration tests in open-source projects. It includes the description of the mined data set (Section 5.1), together with the description of the analysis procedure and the results for RQ 1.1 (Section 5.2.1), RQ 1.2 (Section 5.3.1), and RQ 1.3 (Section 5.4.1).

Chapter 6 presents the results of the quantitative analysis of the differences between unit and integration tests. It includes the used data set description, performed analysis procedure, and the results for RQ 2.1 (Section 6.1), RQ 2.2 (Section 6.2), and RQ 2.3 (Section 6.3).

Chapter 7 presents the results for the qualitative analysis of the differences between unit and integration tests. It includes the analysis of the research and practical view on the topics of test execution automation (Section 7.1), test objective (Section 7.2), and test costs (Section 7.3), including the results for RQ 2.4 (Section 7.1.3), RQ 2.5 (Section 7.2.4), and RQ 2.6 (Section 7.3.3).

Chapter 8 presents the discussion of the results from the qualitative and quantitative analysis. We discuss the effects of our results and their implications out of two different perspectives. First, from the perspective of academia and education (Section 8.1) and second, from a practical perspective (Section 8.2). Moreover, we present threats to the validity of our analysis together with our validation procedures (Section 8.3).

Chapter 9 concludes this thesis with a short summary and an outlook on future work.

2. Foundations

This chapter introduces the foundations of this thesis consisting of different terminology and basic concepts. Section 2.1 presents the concept of software testing together with its related terms. Section 2.2 introduces the concept of mutation testing. In Section 2.3, we present the concept of statistical hypothesis testing together with several tests used within this thesis.

2.1. Software Testing

Software testing is an important aspect for the quality assurance of software. Nowadays, the fundamentals, ideas, and techniques of software testing are essential knowledge for software developers [1]. Sommerville defines software testing as follows [11].

Definition 2.1 (Software Testing)**.** *Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.*

The testing of software can be done on different test levels (Section 2.1.2) and at any time during the software development. Basically, the testing is done by executing a program with artificial data and checking the results of test runs for errors or other anomalies [11]. This testing process has two different goals [11]:

1. demonstrate that the software meets its requirements.
2. discover situations in which the software behaves incorrect, undesirable, or does not conform to its specification.

The first goal is connected to validation testing, which asserts whether a system performs correctly. The second goal is related to defect testing, which tries to expose defects in the software. However, there is no definite boundary between these two approaches [11].

Another important aspect of software testing is often neglected. Software testing cannot show that the software does not contain any defect: there is always the possibility a test exists that discovers further problems within the software. Dijkstra et al. [35] summarize this as follows: “Program testing can be used to show the presence of bugs, but never to show their absence!”.

Software testing is part of the software verification and validation (V&V) process. Within the validation it is asserted if the right product is built, while in the verification it is checked

if the product is built right [36]. The whole V&V process is concerned with evaluating that the developed software meets its requirements and is started as soon as these are available. This process is performed to establish confidence in the built software system [11].

The V&V techniques can be separated into static and dynamic techniques. Static techniques do not need to execute the software for its validation and verification. System requirements, design models, program source code, or test cases itself are typical examples of development artifacts that are validated via static techniques. Two common examples of such techniques are inspections and reviews. On the other hand, dynamic techniques like white-box or black-box testing techniques, must execute the software to verify it [9].

2.1.1. Fundamentals

The IEEE standard ISO/IEC/IEEE 24765-2010 [27] defines the most important vocabulary for the software engineering world. In the following, we present the definitions based on this standard, as well as definitions used by the ISTQB [28], which is a not-for-profit association that provides certification of competences in software testing.

Definition 2.2 (Error)**.** *1. a human action that produces an incorrect result, such as software containing a fault. [...]. [27]*

Definition 2.3 (Fault)**.** *1. a manifestation of an error in software. [...]. [27]*

Definition 2.4 (Failure)**.** *[...] 2. an event in which a system or system component does not perform a required function within specified limits. [27]*

The interconnection between these terms is as follows. A fault (synonym: bug, defect) is a manifestation of an error in a software and may cause a failure. Failures are the observable impact of faults and can be found via software tests responsible for testing a specific software system, the System Under Test (SUT).

Definition 2.5 (Test)**.** *1. an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component [...] 3. a set of one or more test cases and procedures. [27]*

The different tests for a software are organized within test cases. The IEEE defines test cases as follows.

Definition 2.6 (Test Case)**.** *1. a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. [...]. [27]*

Test cases are further organized in test suites. However, the IEEE does not define the term test suite, but includes it into their definition of a test (see above). Hence, we use the definition of the ISTQB, which is as follows.

Definition 2.7 (Test Suite)**.** *A set of test cases or test procedures to be executed in a specific test cycle. [28]*

Definition 2.6 states that the expected result must be defined within a test case. This expected result is determined by the test oracle.

Definition 2.8 (Test Oracle)**.** *A source to determine expected results to compare with the actual result of the SUT. [28]*

The actual result of the execution of a test case against the SUT is compared to the expected result. The *test verdict* associated with this test case is then calculated. The most prominent used verdicts are *pass* and *fail*. Pass is assigned if the actual result and the expected one are equal to each other. Fail is assigned if there is a deviation. If the SUT crashes during the execution of a test case the verdict fail is assigned.

Another important concept in the field of software testing is test coverage. Test coverage measures the completeness of a test suite. Test coverage is defined by the IEEE as follows.

Definition 2.9 (Test Coverage)**.** *1. the degree to which a given test or set of tests addresses all specified requirements for a given system or component. [...] [27]*

There are several test coverage metrics like statement coverage, branch coverage, or Modified Condition/Decision Coverage (MC/DC) coverage. Furthermore, other coverage metrics like requirements coverage or function coverage can be defined [9]. The concrete metric used depends on the SUT, as well as the testing process.

Nowadays, "the testing process usually involves a mixture of manual and automated testing" [11]. In manual testing, a tester generates some test data with which the program is run. She then compares the results to the expected ones and note down deviations. Automated testing runs test cases automatically against the SUT. The comparison of actual and expected results are done via *assertions*. An assertion is a boolean expression which evaluates to false, if the actual and expected results do not match. If this is the case, the test detected a deviation from the expected result and the verdict fail is assigned [11].

2.1.2. Test Level

Testing can be done on several test levels. At each level, tests are based on different software artifacts, e.g., requirements and specifications, design artifacts, or the source code. Each test level accompanies a distinct software development activity [1].

This is visualized in Figure 2.1 showing the general V-Model [37]. It highlights the different test levels and their connection to the development artifacts and their software development activity. Hereby, the constructive activities highlight the typical software development process. At first, the requirements of the software need to be defined and afterwards the functional system design is created. Then the technical system design and component

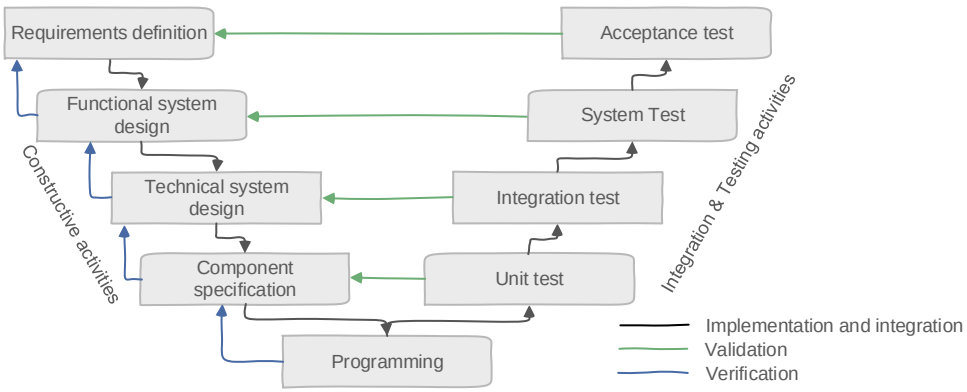


Figure 2.1.: V-Model. Figure adopted from [9].

specification is created before the software can be implemented. At each step of the constructive activities the output of the former phase is verified before the implementation and integration takes place.

The integration and testing activities highlight the different tests that should accompany the constructive activities. Unit tests assess the developed software with respect to its detailed design, as this is the output of the component specification phase. Moreover, integration tests validate the software with respect to its subsystem design. The subsystem design is the output of the technical system design phase. System tests validate the software with respect to its architectural design, which is the output of the functional system design phase. Finally, acceptance tests assess the software with respect to its requirements, which are collected in the requirements definition phase [1].

There are different definitions for the mentioned test levels. Within this section, we present the definitions of the ISTQB and IEEE and highlight their differences for each definition. Both of them are relevant in practice, research, and within this thesis. However, we only describe the first three test levels, i.e., unit test, integration test, and system test, as the acceptance test is not relevant to answer the RQs investigated in this thesis.

A unit is a software artifact that is used as a basis to separate the different test levels from each other. The IEEE and ISTQB define a unit as follows.

Definition 2.10 (IEEE Unit)**.** *1. a separately testable element specified in the design of a computer software component. 2. a logically separable part of a computer program. 3. a software component that is not subdivided into other components. [...]. [27]*

Definition 2.11 (ISTQB Unit)**.** *A minimal software item that can be tested in isolation. [28]*

As the term "minimal software item" is not further defined, we reuse the definition used in the literature [38, 39]: a minimal software item is the smallest compileable unit (i.e., the smallest software item that can work independently). The IEEE definition of a unit is more elaborated, but they are similar to each other and describe the same concept.

Definition 2.12 (IEEE Unit Test)**.** *[...] 3. test of individual hardware or software units or groups of related units. [27]*

Definition 2.13 (ISTQB Unit Test)**.** *The testing of individual software components. [28]*

For the ISTQB a unit test only considers a single unit[1]. This is in contrast to the IEEE, which state that a *unit test* is testing "groups of related units" [27]. Hence, the IEEE definition allows the testing of multiple related units.

Definition 2.14 (IEEE Integration Test)**.** *1. the progressive linking and testing of programs or modules in order to ensure their proper functioning in the complete system. [27]*

Definition 2.15 (ISTQB Integration Test)**.** *Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. [28]*

Here, the definitions of the IEEE and ISTQB are similar. Both describe the linking of different units with the goal to expose interface defects or defects within the interaction between units.

Definition 2.16 (IEEE System Testing)**.** *1. testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. [27]*

Definition 2.17 (ISTQB System Test)**.** *Testing an integrated system to verify that it meets specified requirements. [28]*

Both definitions describe the same concept. System testing is conducted on the whole integrated system to verify if the requirements of the software system are met.

2.2. Mutation Testing

Mutation testing is a technique that is frequently used nowadays [40]. In the following, we summarize the fundamentals of mutation testing (Section 2.2.1) and describe the mutation testing process (Section 2.2.2).

2.2.1. Fundamentals

There are several definitions that are important in order to understand the essence of mutation testing. In the following, we introduce these definitions based on the description by Papadakis et al. [40].

Definition 2.18 (Mutation Analysis)**.** *Mutation analysis refers to the process of automatically mutating the program syntax with the aim of producing semantic program variants, i.e., generating artificial defects. [40]*

[1] Unit is a synonym to component.

Mutation analysis is used within the process of mutation testing to quantify the strength of a test suite [40]. During mutation testing, several artificial defects are integrated into the program code, which are also called mutants.

Definition 2.19 (Mutant)**.** *Semantic program variant with defects. [40]*

However, not all mutants that are created can also be used within the mutation testing process. Mutants that are syntactically illegal, e.g. not compileable mutants, are called stillborn mutants [40] and can not be used.

In a testing context, the mutants must be detected by distinguishing the behavior of the program with the integrated mutant from the original program. If the test is successful in this, the mutant is called *killed* or *detected* and *live* otherwise.

However, there can be different conditions to kill a mutant. Typically, all program outputs are observed for each test run. Hence, everything that the program outputs or asserts is tracked. Using this information we can differentiate between weakly, firm, and strongly killed mutants.

Definition 2.20 (Weakly Killed Mutant)**.** *A mutant is said to be killed weakly, if the program state immediately after the execution of the mutant differs from the one that corresponds to the original program. [40]*

Definition 2.21 (Firm Killed Mutant)**.** *A mutant is said to be killed firm, if the program state comparison at a later point after the execution of the mutant differs from the one that corresponds to the original program. [40]*

Definition 2.22 (Strongly Killed Mutant)**.** *A mutant is strongly killed if the original program and the mutant exhibit some observable difference in their outputs. [40]*

For the weak and firm mutation, the program state has to be changed by the mutant to be killed by a test. However, the output does not necessarily need to be affected by this program state change. This is required by strong mutations. Hence, it is expected that weak and firm mutations are less effective than strong mutations as their program state change does not affect the output of the program. Nevertheless, research showed that there is no formal subsumption relation between these three variants [1].

Definition 2.23 (Mutant Operators)**.** *Syntactic transformation rules to alter the syntax of the program. [40]*

Mutants are generated by applying mutation operators. There exist a large number of mutation operators that were created by researchers. Offutt et al. [41] proposed the five-operator set, which is considered as a minimum standard for mutation testing, including operators like the arithmetic mutation operator. While the definition of mutation operators is easy, the definition of *useful* operators is hard, as they do not only need to be defined but also validated by research studies [40].

However, the selection of mutation operators is a complex task [40]. Researchers often select only a subset of them, because not all of them are applicable to all programs and programming languages. Another reason is that mutation testing is computationally expensive. Hence, its scalability is limited [40].

Definition 2.24 (Mutation Score / Mutation Coverage)**.** *Mutation score or mutation coverage is the number of mutants that are killed by the program's test cases divided by the total number of mutants. [40]*

Overall, the mutation score or mutation coverage highlights how adequate the tests are in testing the program. Therefore, it can be seen as an adequacy metric [1]. Such adequacy criteria define the objectives that we want to reach through testing. According to Papadakis et al. the usage of mutation testing as such a test criterion has three advantages: "to point out the elements that should be exercised when designing tests, to provide criteria for terminating testing (when coverage is attained), and to quantify test suite thoroughness (establish confidence)." [40]. However, using mutation scores as adequacy measure assumes that all generated mutants are of equal value. Nevertheless, recent research showed that this is not the case [42].

Definition 2.25 (Equivalent Mutant)**.** *An equivalent mutant forms a functionally equivalent version of the original program. [40]*

Definition 2.26 (Redundant Mutants)**.** *Redundant mutants are killed whenever other mutants are killed. [40]*

Definition 2.27 (Duplicate Mutants)**.** *Duplicate mutants are mutants that are equivalent between them but not with the original program. [40]*

Definition 2.28 (Subsumed Mutants / Joint Mutants)**.** *Subsumed mutants (or joint mutants) are mutants that are jointly killed when other mutants are killed. [40]*

Duplicate and subsumed mutants are subcategories that belong to the class of redundant mutants. The problem with redundant mutants is that they do not contribute to the test process. Hence, eliminating these mutants only improves the mutation score, but not the selection or generation of test cases. Therefore, the mutation score (Definition 2.24) is inflated and cannot easily be interpreted. Unfortunately, the identification of equivalent and redundant mutants is an undecidable problem [1, 43].

2.2.2. Process

This section presents the different steps of the mutation testing process, as defined by Papadakis et al. [40]. Figure 2.2 pictures a detailed view of this process. The figure presents steps that can be automated in normal boxes, while steps that are inherently manual with boxes that have double lines (i.e., define threshold, P(T) correct, and fix p). Furthermore,

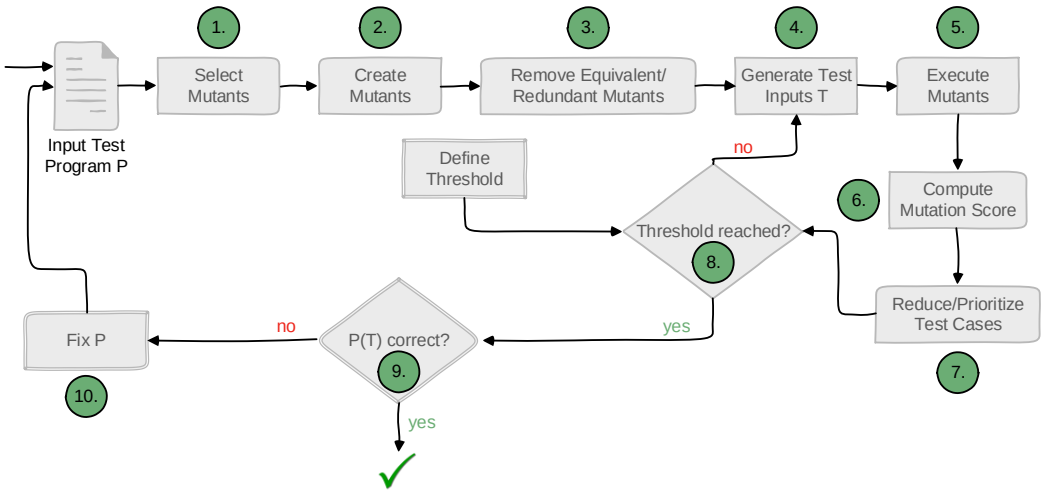


Figure 2.2.: Modern mutation testing process. Boxes with double lines represent steps where human intervention is mandatory. Figure adopted from [40].

the circles highlight the different steps within the process, which we reference to in our explanation below.

1. **Select Mutants:** We have a test program P as input on which we want to apply mutation testing. Therefore, the first step is the selection of mutants. Hence, we select the mutation operators that we want to apply to the program. There exist a lot of different mutation operators, which target different programming languages [44, 45, 46, 47], application types (e.g., Android apps [48, 49] or spreadsheets [50]), types of defects [51, 52, 53], programming elements [54, 55], or others [56, 57]. In addition, the selection of mutants comprises not only of the selection of the mutation operators, but also the selection of mutant reduction strategies, as the number of potential mutants can be enormous. Hence, the selection of a representative subset of mutants is an important step. Prominent techniques include the random selection of mutants [58, 59, 60] and the mutant selection based on their type [46, 61, 62, 63].

2. **Create Mutants:** After the mutants are selected we instantiate them by creating executable programs. The most straight forward way is the creation of a source file for each mutant. However, the costs for creating mutants this way can be high. Research showed, that it takes approximately 3 seconds to compile a single mutant [43]. Hence, different techniques were developed to tackle this problem, e.g., meta-mutation (or mutant schemata) [64, 65] in which all mutants are encoded in a single file, bytecode manipulation [66, 67] in which the bytecode is directly manipulated instead of compiling each single mutant, or the use of interpreted systems [68, 69].

3. **Remove Equivalent / Redundant Mutants:** After the mutants are created, problematic mutants, i.e., equivalent and redundant mutants, need to be removed. However, this is a well-known undecidable problem [70]. Different heuristics were developed by researchers to tackle this problem, e.g., heuristics that rely on compiler optimization techniques [43, 71] or on data-flow patterns [72, 73].

4. **Generate Test Inputs T:** After we have formed our set of mutants we need to generate our test suite against which the mutants should run. This can be done manually or automatically. After the test suite is executed against our set of mutants we determine the mutation score. The test cases of the test suite should be designed in a way that they "reach the mutant, cause an infection on the program state, [and] manifest the infection to the program output" [40]. There exist three different categories of approaches that tackle this problem. First, constraint-bases test generation [68, 74], in which the problem of killing mutants is converted to constraints which need to be satisfied [40]. Second, search-based test generation [75, 76], in which a fitness function is designed that captures the conditions mentioned above to generate test cases. Third, concolic/dynamic symbolic execution [65, 76], which "approximate the symbolic constraints based on the actual program execution." [40].

5. **Execute Mutants:** After the mutants and tests are selected, the mutants get executed. This is the most expensive part of mutation testing [40]. Here, each selected test case is executed against each selected mutant, which can be very time-consuming. Therefore, several approaches were published that address this problem. Overall, if one is interested in the mutation score only, the mutants just need to be executed till some test case kills the mutant. Other approaches try to use this fact by proposing techniques that prioritize such test cases [77] or use a fastest test case first approach [78].

6. **Compute Mutation Score:** After we have determined, how many mutants were killed and lived through the execution of our test suite, we can assess how well the test suite fit by calculating the mutation score. However, the program output needs to be observed to determine which mutants were killed. Therefore, different definitions of a program output can be used (e.g., return values, global variables, thrown errors) to define different killing conditions. The observation is usually done by the test driver, which reports if a test case failed on the mutants during its execution. As mentioned above, equivalent mutants can inflate the mutation score. Hence, researchers developed approaches to approximate the mutation score [79, 80, 81, 82].

7. **Reduce/Prioritize Test Cases:** After the mutation score is calculated, we know how well our test suite performed on the selected mutations. Therefore, we can exclude inefficient test cases from the test suite or prioritize test cases that were most effective. There are several approaches that perform a mutation-based test suite reduction, e.g.,

using a greedy algorithm [83]. The test case prioritization can be done, e.g., based on the number of killed mutants [84, 85] or the distribution of them [84].

8. **Check if Threshold reached:** All steps from 4 to 7 can be repeated until a mutation score is reached which fits the defined threshold. However, this threshold must be set manually. Few research exists that give a concrete answer to the question how the threshold should be chosen. However, recent studies showed that only a high mutation score correlates with the defect detection capabilities of tests [86, 87].

9. **Check results of the test suite execution T on program P:** If we have reached our defined threshold, we need to assess if the results of the test executions were as expected. Therefore, the tester needs to assert the test executions, e.g., by equipping the tests with test oracles. This way, actual defects can be found. While some research exists that try to automate this step (partially) like [75], this step needs to be carried out (mostly) manually.

10. **Fix P:** If defects were found in the program due to the mutation testing, the developers need to fix the program P and repeat the whole mutation testing process till the mutation score threshold is reached again and no defects were found anymore. However, determining the code that is responsible for a defect and the subsequent fixing of the defect must be done (mostly) manually. There are approaches that try to help the developer in localizing the defect, e.g., by the Metallaxis method [88, 89, 90] which is based on the idea "that mutants killed mostly by failing tests have a connection (interaction) with the program defects that caused the program failures" [40]. Moreover, several approaches were developed that support program fixing activities, e.g., by using generated mutations as patch candidates [91, 92].

2.3. Statistical Hypothesis Testing

Statistical hypothesis tests are used frequently in the field of empirical software engineering. In Section 2.3.1, we explain the fundamentals of statistical hypothesis testing, including the most important definitions. Afterwards, in Section 2.3.2, we highlight the typical process of hypothesis testing. In addition, we describe the concept of decision errors (Section 2.3.3) and one-tailed and two-tailed tests (Section 2.3.4). In Section 2.3.5, we explain concrete statistical hypothesis tests that are used in this thesis. Afterwards, in Section 2.3.6, we describe the effect size together with one concrete realization. Finally, in Section 2.3.7, we explain the multiple comparison problem together with one possible solution, which is the Bonferroni correction.

2.3.1. Fundamentals

Two concepts that are important to understand statistical hypothesis tests are the concept of a population and a sample. Both of these concepts are defined below.

Definition 2.29 (Population)**.** *The population is the total set of observations that can be made. [93]*

Definition 2.30 (Sample)**.** *A sample is one or more observations drawn from the population. [93]*

Statistical hypothesis tests are part of the area of inferential statistics, which belong to the area of statistics [94]. The goal of inferential statistics is to calculate different parameters of the sample (e.g., mean or variance) and infer these parameters for the whole population from the calculated ones [94]. For most questions, the collection of data for the whole population is impossible or too costly. For example, if we want to evaluate the mean of the weight of people living in Germany, we would need to collect data from over 80 million people, which would cost time and money [94]. However, inferential statistics shows that it is possible to estimate parameters like the mean for a whole population based on a sufficient large sample.

Definition 2.31 (Statistical Hypothesis)**.** *A statistical hypothesis is an assumption about a population parameter. [93]*

Definition 2.32 (Hypothesis Testing)**.** *Hypothesis testing refers to the formal procedures used by statisticians to accept or reject statistical hypotheses. [93]*

Therefore, we generate a statistical hypothesis regarding a parameter of a population based on our sample and test this hypothesis via statistical hypothesis testing. The result shows whether the stated statistical hypothesis is to be accepted or rejected. There exist two different types of statistical hypothesis, which are defined below.

Definition 2.33 (Null Hypothesis)**.** *The null hypothesis, denoted by H_0, is usually the hypothesis that sample observations result purely from chance. [93]*

Definition 2.34 (Alternative Hypothesis)**.** *The alternative hypothesis, denoted by H_1 or H_a, is the hypothesis that sample observations are influenced by some non-random cause. [93]*

To illuminate these two types of hypothesis we give a short example. Consider that we want to determine the fairness of a coin. Our null hypothesis could be that half of the coin flips would result in Heads and the other half in Tails, as this would be the "normal" outcome for a fair coin. Our alternative hypothesis in this case is that both (the number of Heads and Tails) are different. Suppose that we flipped the coin 100 times (our sample) and these coin flips resulted in 80 Heads and 20 Tails. Based on our sample we would conclude that the coin is probably not fair, because there is a difference between the number of Heads and Tails [93].

2.3.2. Process

Hypothesis testing is performed to decide, whether we should reject or fail to reject the null hypothesis[2]. The basic hypothesis testing process consists of four steps, which are explained below [96].

1. **State the hypotheses:** We need to formulate the problem at hand in terms of hypotheses. This involves the creation of a null and an alternative hypothesis. They must be mutually exclusive. Neave [97] states that the researcher should first concentrate on the alternative hypothesis, because this hypothesis is more important practically.

2. **Formulate an analysis plan:** The analysis plan should specify two elements: the significance level (often abbreviated with α) and the test method. Usually, significance levels equal to 0.001, 0.01, 0.05, or 0.10 are chosen [94]. However, recent research suggests to use a significance level of 0.005 [98]. The test method includes a test statistic (e.g., mean score, z-score, t statistic, chi-square) and a sampling distribution. The test statistic is computed from the sample data. The sampling distribution is the probability distribution of a statistic from all possible samples of size n drawn from a given population [93]. If the test statistic and its sampling distribution is given, we can assess the probability of the test statistic. If this probability is lower than the chosen α, we reject the null hypothesis [93].

3. **Analyze sample data:** As a next step, we use our collected sample data to perform the computations described in the analysis plan. Hence, we compute the test statistic, potentially including the standard deviation or standard error of the statistic. Additionally, we calculate the p-value which "is the probability of observing a sample statistic as extreme as the test statistic, assuming the null hypothesis is true." [93].

4. **Interpret the results:** Finally, we interpret our results. Typically, we compare the calculated p-value to our chosen significance level α. If the p-value is smaller than our significance level, we reject the null hypothesis and fail to reject it otherwise.

2.3.3. Decision Errors

If we perform a hypothesis test two different types of errors could occur: Type I errors (also called α-error) and Type II errors (also called β-error). Both of them are explained below.

- **Type I error:** Type I errors occur, if we reject the null hypothesis when it is true. The probability of committing such an error is also called significance level and should be chosen when an analysis plan is formulated (see above) [94].

[2] Statisticians believe that if we would state that we accept the null hypothesis this would imply that the null hypothesis is true, which is a fallacy. Therefore, they propose to state that we "fail to reject" the null hypothesis as this implies that the data basis is not sufficiently persuasive to chose the alternative hypothesis over the null hypothesis. [95]

- **Type II error:** Type II errors occur, if we fail to reject the null hypothesis when it is false. The probability of **not** committing such an error is called the power of the test [94].

2.3.4. One-Tailed and Two-Tailed Tests

Statistical hypothesis can be directed and undirected. Undirected hypothesis state that there is a connection (or difference) between two characteristics, but do not state the direction of the connection (or difference).

One example for an undirected hypothesis would be: "The educational background differs on the gender". A directed hypothesis include a direction in which a connection is assumed. One example for such a hypothesis would be: "Women have a higher educational background than men" [94]. Directed hypothesis are tested via one-tailed statistical tests, while undirected hypothesis are tested via two-tailed tests.

Definition 2.35 (One-Tailed Test)**.** *A test of a statistical hypothesis, where the region of rejection is on only one side of the sampling distribution, is called a one-tailed test. [93]*

Definition 2.36 (Two-Tailed Test)**.** *A test of a statistical hypothesis, where the region of rejection is on both sides of the sampling distribution, is called a two-tailed test. [93]*

2.3.5. Concrete Statistical Hypothesis Tests

Within this section several concrete statistical hypothesis tests are explained. Shapiro-Wilk, which is a test for normality is described in Section 2.3.5.1. Afterwards, in Section 2.3.5.2 the Brown-Forsythe test is explained, which tests for equal variances between two populations. In Section 2.3.5.3 the t-test is explained. Finally, the Mann-Whitney-U test is described in Section 2.3.5.4.

Within these sections, we also explain how the different test statistics are calculated. A test statistic is a random variable, which is calculated from the sample data. Basically, the test statistic compares our sample data with what is expected under the null hypothesis of the specific statistical test. For a normality test like the Shapiro-Wilk test this means, that the test statistic gives a measure of the degree of agreement between the sample data and the hypothesis that the population from which this sample data is drawn from follows a normal distribution. When the degree of agreement is low, the test statistic becomes large (or small, depending on the alternative hypothesis). This results in a small p-value, which will then lead to the rejection of the null hypothesis [99].

2.3.5.1. Shapiro-Wilk Test

The Shapiro-Wilk test [100] was first published by Samuel Shapiro and Martin Wilk in 1965. It is a statistical hypothesis test, which tests if the population from which a sample was drawn follows a normal distribution.

The H_0 hypothesis is that the population follows a normal distribution, while the H_1 states the opposite. Hence, if the p-value is below the chosen significance level the H_0 must be rejected. Therefore, we can state that the population does not follow a normal distribution. Otherwise, if the p-value is above the chosen significance level and we fail to reject H_0 we can state that the population follows a normal distribution [100].

The Shapiro-Wilk Test has several requirements that need to be fulfilled to be applied:

1. the observations of the sample must be independent of each other,
2. the sample must be between 3 and 5000 observations,
3. the underlying random variable of the observations must inherit a metric scale.

However, in contrast to other tests for normality, like the Kolmogorov-Smirnov-Test [101] or the Chi-Square-Test [102], the Shapiro-Wilk test is efficient especially for samples with a sample size < 50 [100].

The test statistic is defined as follows

$$W = \frac{(\sum_{i=1}^{n} a_i x_{(i)})^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}, \tag{2.3.1}$$

where

- $x_{(i)}$ is the ith smallest number in the sample;
- $\bar{x} = (x_1 + ... + x_n)/n$ is the sample mean;
- the constants a_i are given by

$$(a_1, ..., a_n) = \frac{m^T V^{-1}}{(m^T V^{-1} V^{-1} m)^{1/2}}, \tag{2.3.2}$$

  where

$$m = (m_1, ..., m_n)^T \tag{2.3.3}$$

  and $m_1, ..., m_n$ are the "expected values [...] of the standard normal order statistics" [100].

The test statistic has two different parts. The estimator in the numerator of the W-statistic ($(\sum_{i=1}^{n} a_i x_{(i)})^2$) calculates how the variance of a sample has to be when drawn from a population that follows a normal distribution. Another estimator in the denominator of the W-statistic ($\sum_{i=1}^{n}(x_i - \bar{x})^2$) calculates the variance of the sample. These two parts are then compared against each other to assess if both estimators come to the same result. The closer both estimations are to each other the more likely it is that the population of the sample follows a normal distribution.

2.3.5.2. Brown-Forsythe Test

The Brown-Forsythe test was first published by Morton Brown and Alan Forsythe in 1974 [103]. It is similar to the Levene test [104] with the difference that instead of assessing the means of the different samples to test if two (or more) populations possess equal variances (homoscedasticity) the Brown-Forsythe test uses the median.

The H_0 hypothesis of the Brown-Forsythe test states that all population variances are equal (i.e., $H_0 : \sigma_1^2 = \sigma_2^2 = ... = \sigma_k^2$). The H_1 states that at least one pair of variances are different (i.e., $H_1 : \sigma_i^2 \neq \sigma_j^2$ for at least one pair i, j with $i \neq j$). Hence, if the p-value of the Brown-Forsythe test is below the chosen significance level, we can reject H_0. Hence, the different populations do not have equal variances (heteroscedasticity). Otherwise, we fail to reject H_0, which means that all populations are homoscedastic.

The test statistic is defined as follows

$$F = \frac{(N-k)}{(k-1)} \frac{\sum_{i=1}^{k} N_i (Z_{i.} - Z_{..})^2}{\sum_{i=1}^{k} \sum_{j=1}^{N_i} (Z_{ij} - Z_{i.})^2}, \tag{2.3.4}$$

where

- k is the number of different samples to which the observations belong,
- N_i is the number of observations in the ith sample,
- N is the total number of observations in all groups,
- Y_{ij} is the value of the measured variable for the jth observation from the ith sample,
- $Z_{ij} = |Y_{ij} - \tilde{Y_{i.}}|, \tilde{Y_{i.}}$ is the median of the ith sample,
- $Z_{i.} = \frac{1}{N_i} \sum_{j=1}^{N_i} Z_{ij}$ is the mean of the Z_{ij} for sample i,
- $Z_{..} = \frac{1}{N} \sum_{i=1}^{k} \sum_{j=1}^{N_i} Z_{ij}$ is the mean of all Z_{ij}.

There exist some critic on this tests and especially on the original Levene test, as the means of the different samples could be biased. The Brown-Forsythe test eases this threat by using the median instead of the mean. While Olejnik and Algina [105] showed that the Brown-Forsythe test is very precise, even if the distribution of the populations are not normal distributed. However, Glass and Hopkins [106] raised critic on the Levene and Brown-Forsythe tests. They state that both of these tests have a “fatal flaw” which is the assumption that the deviations from the mean (or median) of the different samples have an effect on the homoscedasticity.

2.3.5.3. T-Test

The t-test is a statistical hypothesis test that was first developed by Student [107] in 1908. It is used to determine, if the means of two populations are equal or different. It can be used to evaluate if two data sets are significantly different from each other [107].

The t-test has several shapes. The t-test can have one or two samples. The one-sample t-test evaluates whether the mean of a single population is equal to an unknown mean. The two-sample t-test can be applied to independent (or unpaired) samples, as well as paired samples. The independent samples t-test compares the means of two populations, while the paired samples t-test compares the means from the same population, but at different times. However, within this thesis we only make use of the two-sample t-test using independent samples with equal sample sizes. Therefore, only this t-test is explained within this section.

The t-test has several assumptions that must be fulfilled before it can be applied. These assumptions are listed below.

- Each of the two populations that are compared follow a normal distribution. This can be tested, e.g., via the Shapiro-Wilk test (Section 2.3.5.1).
- The two populations that are compared should have the same variance. This can be tested, e.g., via the Brown-Forsythe test (Section 2.3.5.2). If this assumption is not met by the data, a variant of the t-test, i.e. the Welch t-test [108], can be used.
- Both samples should be independent from each other.

Basically, the t-test evaluates if the means of the two populations from which the two samples are drawn are different. For this, it is using the means of both samples. Therefore, the H_0 for this kind of t-test is that both population means are equal (i.e., $H_0 : \mu_1 = \mu_2$), while the H_1 states the opposite (i.e., $H_1 : \mu_1 \neq \mu_2$). However, the t-test also has a one-sided version, in which the H_0 is that the mean of the first population is greater than or equal (or less than or equal) than the second mean (i.e., $H_0 : \mu_1 >= \mu_2$ or $H_0 : \mu_1 <= \mu_2$) while the H_1 states that the mean of the first population is smaller (or greater) than the mean of the first population (i.e., $H_1 : \mu_1 < \mu_2$ or $H_1 : \mu_1 > \mu_2$). If the p-value of the t-test is below the chosen significance level, we can reject H_0. Hence, the mean of both populations are different.

The t-statistic for independent samples with equal sample size is calculated as follows:

$$t = \frac{\bar{X_1} - \bar{X_2}}{s_p \sqrt{\frac{2}{n}}} \tag{2.3.5}$$

where

- $\bar{X_1}$ and $\bar{X_2}$ are the means of the two samples,

- $s_p = \sqrt{\frac{s_{X_1}^2 + s_{X_2}^2}{2}}$, which is the weighted standard deviation and $s_{X_1}^2$ and $s_{X_2}^2$ are the variances of the two samples,
- n is the number of observations in each sample.

For the Welch t-test, the t-statistic is calculated differently. The denominator is not based on the weighted standard deviation in this case. Hence, the t-statistic for the Welch t-test is calculated as follows.

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{n} + \frac{s_2^2}{n}}} \tag{2.3.6}$$

where

- $\bar{X}_1$ and $\bar{X}_2$ are the means of the two samples,
- s_1^2 and s_2^2 are the sample standard deviations,
- n is the number of observations in each sample.

2.3.5.4. Mann-Whitney-U Test

The Mann-Whitney-U test (or Mann-Whitney-Wilcoxon, Wilcoxon-Mann-Whitney test, U-test, or Wilcoxon rank-sum test) is a non-parametric test that was developed by Mann and Whitney [109] and Wilcoxon [110]. This test is used to determine if two distributions are based on the same distribution.

The Mann-Whitney-U test makes several assumptions that must be checked before it can be applied. These assumptions are listed below.

1. All observations from both samples are independent from each other.
2. It is possible to order the observations (i.e., for every two observations, it is defined which of the two is greater).
3. The underlying populations from which both samples are drawn must be homoscedastic. This can be tested, e.g., with the Brown-Forsythe test (Section 2.3.5.2).

In a first step, all observations from both samples are combined, ordered by value, and assigned ranks to reflect their position in a list. If there is a tie in this list (i.e., two observations have the same value), an average of their rank values is taken. After the list is complete and all ranks have been assigned, the U statistic is computed for both samples as follows [109].

$$U_1 = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1, U_2 = n_1 n_2 + \frac{n_2(n_2+1)}{2} - R_2, \tag{2.3.7}$$

where

- n_1 is the number of observations in the first sample,
- n_2 is the number of observations in the second sample,
- R_1 is the sum of ranks for the observations of the first sample,
- R_2 is the sum of ranks for the observations of the second sample.

The resulting U-statistic is the minimum of U_1 and U_2. The Mann-Whitney-U test can be one or two-sided. For the two-sided version, the H_0 states that there is no difference between the ranks of both samples, while the H_1 states the opposite (i.e., there is a difference between the ranks of both samples). In the one-sided version the H_0 states that the ranks of first sample are greater than or equal (or less than or equal) than the ones of the second sample, while the H_1 hypothesis states the opposite (i.e., the ranks of the first sample are less (or greater) than the ranks of the second sample). The H_0 hypothesis will be rejected, if the p-value is below the critical significance level [109].

In contrast to the t-test (Section 2.3.5.3) the Mann-Whitney-U does not need the assumption that both populations from which the samples are drawn are normally distributed, as it is a non-parametric test. This allows us to check for statistically significant differences between two populations, even if they do not follow a normal distribution.

2.3.6. Effect Size and Cohen's d

The effect size is a metric that is used to evaluate the strength of a statistical claim and "is the difference between the true value and the value specified by [the] null hypothesis" [111]. It has an influence on the power of the test, which is the probability of not committing a Type II error (or β-error) [111]. Cohen's d is a formula to calculate the effect size of a statistical hypothesis test. It was first published by Cohen in 1977 [112].

Cohen's d calculates the difference between two means and divide it by the standard deviation of the data. The formulas and explanations are shown below [112].

$$d = \frac{\bar{x_1} - \bar{x_2}}{s}, \tag{2.3.8}$$

where

- $\bar{x_1}$ is the mean of the first sample,
- $\bar{x_2}$ is the mean of the second sample,
- $s = \sqrt{\frac{(n_1-1)s_1^2+(n_2-1)s_2^2}{n_1+n_2-2}}$, where
  - n_1 is the number of observations in the first sample,
  - n_2 is the number of observations in the second sample,
  - $s_1^2 = \frac{1}{n_1-1}\sum_{i=1}^{n_1}(x_{1,i} - \bar{x_1})^2$, which is the variance of the first sample,

Effect Size	d
Very small	$d(.01)$
Small	$d(.2)$
Medium	$d(.5)$
Large	$d(.8)$
Very large	$d(1.2)$
Huge	$d(2.0)$

Table 2.1.: Rules of thumb for effect sizes. Based on [113].

- $s_2^2 = \frac{1}{n_2-1}\sum_{i=1}^{n_2}(x_{2,i} - \bar{x_2})^2$, which is the variance of the second sample.

The resulting d is a one dimensional number. However, its interpretation is not defined, because it is only a number. Cohen gives a rule of thumb for its interpretation in his paper [112]. These rules of thumb were later extended by Sawilowsky [113]. Table 2.1 highlights the extended rules of thumb, as proposed by Cohen and Sawilowsky [112, 113].

2.3.7. Multiple Comparison Problem and Bonferroni Correction

Multiple comparisons using the same data can become a problem, as the likelihood of rejecting the H_0 incorrectly (i.e., Type I error) increases [114]. The Bonferroni correction is a method to counter the problem of multiple comparisons in statistical hypothesis testing [115]. It counters this problem by testing each individual hypothesis that is done on the same data at a significance level of $\frac{\alpha}{n}$, where α is the overall desired significance level and n is the number of hypotheses that we test.

The argument for the Bonferroni correction is as follows [115]. Let $H_1, ..., H_n$ be the statistical hypothesis and $p_1, ..., p_n$ their corresponding p-values. Let n be the number of null hypotheses and n_0 the number of true null hypotheses. As stated above, using the Bonferroni correction we reject the null hypothesis for each $p_i \leq \frac{\alpha}{n}$. Hence, we control for the Family-wise Error Rate (FWER), which is the probability of making Type I errors when performing more than one hypotheses tests.

Theorem 2.1 (Bonferroni Inequality)**.** *For a countable set of events $A_1, A_2, A_3, ...$, the following inequality equation holds true: $P(\cup_i A_i) \leq \sum_i P(A_i)$.*

Proof. We can proof that the FWER using the Bonferroni correction is equal to the chosen significance level α, as follows [116]:

$FWER = P\{\cup_{i=1}^{n_0}(p_i \leq \frac{\alpha}{n})\} \leq \sum_{i=1}^{n_0}\{P(p_i \leq \frac{\alpha}{n})\} \leq n_0\frac{\alpha}{n} \leq \alpha$ □

However, applying the Bonferroni correction or any FWER control has a cost. It increases the Type II error (i.e., the production of false negatives) and therefore reduces the statistical power of the tests [117].

3. Related Work

This thesis describes an empirical study on the differences between unit and integration tests. We need to acquire different data to perform this study to evaluate all differences mentioned in Section 1.1. This includes the classification of tests into unit and integration tests, the assessment of their effectiveness including their effectiveness per defect type, and the evaluation of the defect-locality. Hence, we start by discussing the related work to each of these fields within this section. Afterwards, we discuss related work that have a direct connection to our RQs, i.e., the distribution of unit and integration tests in software projects and evaluations of differences between unit and integration tests. In the end, we give a short summary of the identified related work and define our research delta.

3.1. Test Level Classification

There are several approaches present in the literature that classify tests into different test levels. However, they differ in the data that is used for the classification, the used test level, and the process of the classification.

One of the most recent works is the short paper by Orellana et al. [118]. The authors of this paper differentiate unit and integration tests based on the build process of the project at hand. If a test class is executed by the Maven SureFire plugin [119], they are classified as unit tests. All tests that are executed by the Maven FailSafe plugin [120] are classified as integration tests. However, this classification process can be problematic due to several reasons. First, our work [38] highlights that the classification into those two test levels, as done by the developers, is not always in line with the definitions of the IEEE or ISTQB. Within this work we determine how many tests are unit tests, according to the definitions of the ISTQB and IEEE for 10 Python projects. We use a static analysis approach in which we assess the number of imported modules in a test to find unit tests. Afterwards, we compare the developer classification of tests with the classification according to the definitions. There we found, that developers are not classifying their tests in accordance to the ISTQB or IEEE definitions. Second, the analysis of Orellana et al. [118] is on a rather coarse-grained level, as whole test classes are classified instead of test methods. Third, the number of projects to which the classification schema can be applied is rather limited, as these projects would need to use both: the Maven SureFire and the Maven FailSafe plugin. For none of the projects that we have used in our study is this the case.

In contrast to the work of Orellana et al. [118], our approach neither needs the projects build file nor makes it assumption on the used build system. Instead, we only need the coverage data of a test execution run. In addition, our approach is more fine-grained than the approach by Orellana et al. [118], as we classify test methods instead of a whole test class or module. This better reflects the testing reality and allows a finer-grained analysis.

One of the first approaches that classify tests into different test levels was proposed by Kanstrén [121]. He proposes a dynamic aspect oriented programming based approach that calculates the "test granularity" for each test. The test granularity refers "to the number of units of production code included in a test case" [121]. Kanstrén's approach calculates this metric by summing up the number of methods that are covered by each test. Afterwards, the results can be summarized (e.g., within a bar plot) to determine if, e.g., a program was tested only on low-level (i.e., many tests that executed a small amount of methods) or only on high-level. However, Kanstrén does not provide a clear separation criterion to differentiate the tests.

Similar to Kanstrén [121], our own classification approach is also dynamic, i.e., we need to execute the tests before we can classify them. This has several advantages. For example, we can directly determine if a unit was used within a test (and not only imported). Furthermore, a dynamic approach is robust against modern techniques like reflection [122] or the usage of mocking frameworks, i.e., our classification is not influenced by it. In contrast to the work of Kanstrén [121] our approach makes a clear separation between unit and integration tests based on common definitions. However, Kanstrén [121] only provides a survey of the granularity of tests within a project.

3.2. Test Effectiveness Assessment

There are different approaches to assess the effectiveness of tests. They all fall into the category of fault-based testing [123]. In fault-based testing, artificial defects are introduced into the program. Afterwards, the test suite is executed to determine if the test cases are able to detect the integrated defect. The major challenge of this approach is the seeding of defects. Ideally, the seeded defects should be representatives of real life defects [124]. Otherwise, the results of the analysis do not represent the effectiveness of the tests in a real life setting. The literature discusses several solutions to this problem, like random seeding [125], the seeding of defects based on the program dependence graph [126], or the hand-seeding of defects [127]. In our work, we use another approach which is called mutation testing [124] on which we focus in the following.

The field of mutation testing is large and a lot of contributions were done in the past years. Mutation testing is used nowadays to, e.g., automatically repair software programs [128], automatically localize defects [88], or automatically improve non-functional properties of programs like security [51], execution speed [129, 130], or memory consumption [129]. The importance and large number of papers in this field is also reflected by the number of

literature surveys that exist, e.g., by Offutt and Untch [131], Jia and Harman [132], and Papadakis et al. [40].

The use of mutation testing to assess the effectiveness of tests is common in the current research (e.g., [133, 134]). It is increasingly used as a fundamental experimental methodology, as Papadakis et al. [40] highlight. However, its use to assess the defect detection capabilities of tests is controversial. It has the underlying assumption that the mutants can construct program failures that are similar to the ones that are created through real defects. Several studies provide support for this assumption. Daran et al. [135] did one of the first studies that investigated the relationship between real defects and mutants. Their results show that mutants and real defects can produce similar erroneous program states. Andrews et al. [136, 137] came to a similar result: they conclude that the detection ratio of mutants is representative for the the defect detection ratios. In one of the most recent papers Just et al. [138] highlight, that there is a strong correlation between the real defect detection ratios and mutant detection ratios.

Nevertheless, some recent papers identified limitations to the results presented above. Namin et al. [139] found only a weak correlation between the defect detection ratios and their injected mutants. Chekam et al. [86] found a strong correlation between defect detection and the increase of the mutation score, which is the quotient from the number of detected mutants and the total number of mutants. Nevertheless, they were only able to identify this correlation for higher mutation score levels. The most recent paper by Papadakis et al. [87] found only a weak correlation between defect detection and mutation scores, if the size of the test suite is controlled for in experiments. The different studies performed by different authors highlight that there exist no definite answer to the question if mutation testing is an appropriate tool to assess the defect detection capabilities of tests. While we also make use of mutation testing in this thesis, we do not create test suites by ourselves. Instead, we reuse the test suites provided by the developers of the projects. Hence, some limitations mentioned above, e.g., that the test suite size must be controlled for, are not applicable to our research.

3.3. Defect Classification

There are numerous studies on the classification of defects. They mainly differ in the data on which the classification is based. Some taxonomies need software specification or design documents (e.g., [140]), others need source code (e.g., [141]), or defect reports (e.g., [142], [143]).

The most commonly used cause-driven taxonomy is the Orthogonal Defect Classification (ODC) and was proposed by Chillarege et al. [144]. In ODC, defects can be classified into eight different types. The decision is made based on their description about the symptoms, semantics, and root causes. Another model for the characterization of defects was proposed by Offutt and Hayes [127]. Within their model, defects are classified based on the

syntactic and semantic size. Later, Hayes [140] presented a defect analysis methodology that is based on requirements. In addition, Hayes applied this model to NASA projects. In 2005, Hayes [145] published two more taxonomies, where the first classifies code modules (e.g., view, controller, data-centric,...) and the second code defects (e.g., into data, interface, computation). Xia et al. [142] proposed a classification of defects based on the description in defect reports. In this classification schema, the defects are classified into two different defect trigger categories (Bohrbug and Mandelbug) via natural language processing technique. Tan et al. [143] also used defect reports and proposed a classification in which a defect is classified based on three different dimensions: the root cause of the defect, the impact (i.e., failure caused by the defect), and the component (location of the defect).

The main disadvantage of the above mentioned approaches is that the needed data to classify the defects (e.g., defect descriptions or design documents) are often not available. This is especially true for open-source projects, as the development of open-source projects is different from the traditional software development process [146]. In addition, the above mentioned taxonomies have the problem that the creation of a link between the classified defect and its source code representation is hard to achieve.

Zhao et al. [141] recently proposed an approach that can overcome the problems described above. They adapted the classification by Hayes et al. [145] and classify defects based on the change that was made to fix it. Zhao et al. [141] created a tool for the C language that is able to calculate the defect class: it gets the defective and clean version as input, calculates the changes between theses versions, and detect different change patterns. Afterwards, the defect gets classified based on these change patterns. Overall, they created five different defect categories, based on the categories by Hayes et al. [145]. They also defined nine different subcategories in which defects can be classified. The classification scheme of Zhao et al. [141] is shown in Figure 3.1. The computation category only includes Changes on Assignment Statements (CAS). Computation-related defects can lead to a wrong assignment of a variable. The data category includes Changes on Data Declaration and Definition (CDDI) statements. Zhao et al. [141] reason, that if the declared type of a variable is changed (e.g., from int to float), a data-related defect occurred. Interface-related defects are caused "by wrong definition or faulty function dependency on other functions." [141]. This includes, e.g., defects where a function is called with an incorrect amount of parameters, or a misplaced function call. These type of defects are then fixed by Changes on Function Declaration/Definition (CFDD) or Changes on Function Call (CFC). Logic/Control defects "may cause the incorrect execution sequence or an abnormal state" [141]. It comprises of Changes on Loop Statements (CLS) (e.g., if the initialization of a for-loop is changed), Changes on Branch Statements (CBS) (e.g., if a $<$ is changed to a $>=$ in an if-statement), and Changes on Return/Goto Statements (CRGS) (e.g., if the return value of a function was changed). All changes that cannot be classified into the categories above are then subsumed in the Others category. Zhao et al. [141] further subdivided this category into Changes on Preprocessor Directives (CPD) and Others (CO). The CPD category comprises of changes that are C-language specific, while the CO category includes

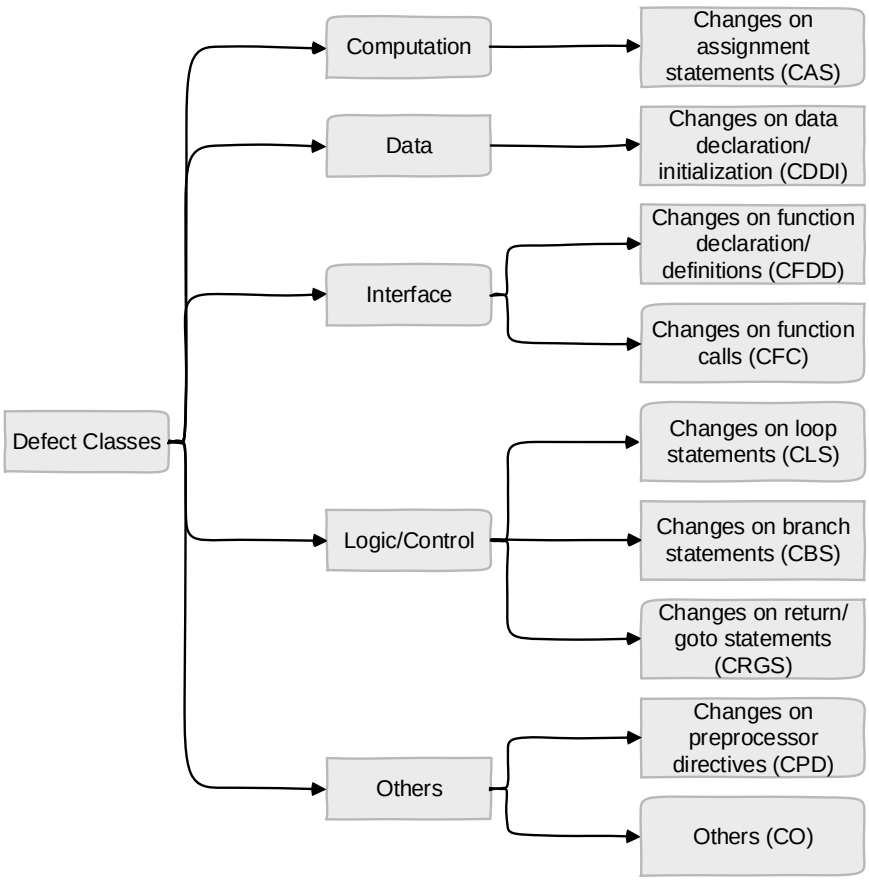


Figure 3.1.: Defect classification by Zhao et al. [141]. Figure adopted from [141].

all other changes. Besides the work of Zhao et al. [141] there are also other approaches that classify source code changes (e.g., [147]). However, these are not further discussed, as they do not provide a defect taxonomy and/or a mapping of source code changes to defect classes.

Our defect classification approach is based on the work of Zhao et al. [141], as we need an approach that only needs the source code for the defect classification. During our study, we generated different mutants which, as they are not regular defects, do not exhibit, e.g., an issue report. We reuse the logic behind every subcategory presented by Zhao et al. [141], but only reuse the main category in which a defect resides in (e.g., “Computation”). Another difference from our work to the work of Zhao et al. [141] is that we applied our defect classification approach to Java projects, instead of C projects. Furthermore, we excluded the CPD sub category as such changes do not occur in Java projects.

3.4. Defect-Locality

In our work, we examine if unit tests are better able to pinpoint the source of defects than integration tests. The goal of debugging is to locate the defect. Normally, this is done by developers by going through the calls till the defect occurred. There exist a lot of different approaches, which try to automate the localization of defects (e.g., [148, 149]), but these works are out of scope of this thesis, as we do not try to locate the defect (as we know its location already). However, we can learn from the related work in this field by evaluating which data is used to make a localization possible and which approaches are used to collect this data.

While there are several types of data used as a basis to locate defects (e.g., history data [150, 151], program spectra [152, 153], or object-usage [154, 155]), we found two data types that are of interest for our approach: call traces and coverage.

Our approach is inspired by the works of Dallmeier et al. [156] and Jones et al. [157]. Dallmeier et al. [156] use call sequences of programs (one correct and one defective one) to localize the defect. They utilize the Java instrumentation API [34] to collect the call traces along the program run on a per-object basis. The authors conducted two different experiments using their implementation and found that (1) call sequences are better than coverage to predict defects, (2) per-object sequences improve the prediction over global sequences, and (3) the caller of a method is more likely to be defective in contrast to the callee.

Jones et al. [157] created a technique that make use of colors to "visually map the participation of each program statement in the outcome of the execution of the program with a test suite, consisting of both passed and failed test cases." [157]. They utilize statement coverage to try to locate a defect in a program. A statement is more likely to contain the defect the more often it is executed in failing test case runs. The authors found in their evaluation that within a program with only one fault this fault is almost certainly marked as "likely faulty". However, 5% to 15% of the correct code is also marked. If programs contain more than one fault, these numbers degrade to 5% to 20%.

In our approach we are mixing the collection of call traces and coverage to gather the defect-locality of a defect for a test. Basically, we record the call traces by using the Java instrumentation API (similar to [156]) of a test and check when our integrated defect was covered. Then, the depth of our call stack is stored in a database for this particular defect and test.

3.5. Distribution of Unit and Integration Tests

The research on the distribution of unit and integration tests is currently in its beginning. While there are several works that had a look at the overall amount of tests [158, 159, 160] or how they evolve (e.g., [161, 162]) there are, to the best of our knowledge, no publications

that have a look at the distribution of unit and integration tests. During our literature study, we only found one blog post in the Google testing blog [163], which states, that "Google often suggests a 70/20/10 split: 70% unit tests, 20% integration tests, and 10% end-to-end tests." [163]. However, there is no evidence given that supports this separation or shows if this really applies to current software projects.

3.6. Differences between Unit and Integration Tests

There are numerous papers on unit and integration testing. These works focus on, e.g., finding links between the test code and the code that is tested [164, 165], finding test smells [22, 23, 166], proposing test refactorings [23, 166], detecting test refactorings [23, 166], visualizing test executions [21], test case minimization [21, 167], and test generation [18, 19, 20]. To the best of our knowledge, there is only the paper of Orellana et al. [118], that compares unit and integration tests with each other.

The authors evaluated if either unit or integration tests detect more defects by using the Travis Torrent data set [168]. They determined the number of defects that were exposed by using data that was contained inside this data set (i.e., which test case failed during a project build). The underlying assumption for this kind of analysis is that every failing test during a project build exposed a real defect. However, this might not be the case. For example, tests in a Continuous Integration (CI) system could also fail due to wrong commit behavior of the developer (e.g., if the developer forgot to commit changes to a class interface). Furthermore, this technique could only assess defects that were detected by the CI system. Hence, defects that might get fixed before the changes are pushed to a CI system are missing.

To overcome the mentioned limitations in the approach by Orellana et al. [118] we decided to use mutation testing to assess the effectiveness of unit and integration tests. Therefore, we create a controlled environment and can assess the potential defect detection capabilities of the tests on each test level in a systematic way.

3.7. Summary and Research Delta

The research deltas that we strive for in this thesis are concerned with the distribution of unit and integration tests, as well as the evaluation of each difference between them that are mentioned in the literature. The analyzed related work highlights that, to the best of our knowledge, only one paper compares one difference between unit and integration tests (i.e., the paper by Orellana et al [118]).

We found no related work concerning the distribution of unit and integration tests in software projects. While there exist works that had a look at the overall amount of tests, there are no specific publications, except a blog post, that specifically look at the distribution of tests on the different test levels. Hence, within this thesis we improve the body of knowl-

edge of software testing by providing an empirical evaluation of the distribution of unit and integration tests in open-source projects.

While there exist some approaches to classify tests into different test levels, we improve the current state of the art by designing a technique that is able to precisely assign a test level to each test method of a test class. This means, we developed a more fine-grained approach than described in the existing literature.

The use of mutation testing to assess the effectiveness of tests is common practice [40]. However, there exist no related work that compares the effectiveness of unit and integration tests with each other in respect to the type of detected defects. The only work in this field is the work by Orellana et al. [118] who compared the overall effectiveness of unit and integration tests by analyzing the number of failed tests in the build logs of projects. Nevertheless, the approach by Orellana et al. [118] is coarse-grained and does not take the type of the found defect into account. Hence, we improve the state of the art by providing an empirical comparison of the test effectiveness of unit and integration tests, separated by defect type.

The analysis of our related work shows that most of the data collection techniques that we use within this thesis are not completely new, but based on related work and improved within this thesis. However, the analysis on the data (i.e., the analysis of the differences between unit and integration tests) is new and broadens the body of knowledge of software testing research.

4. Research Methodology

In this chapter, we describe the research methodology for our study. At first, we give an overview of our study in Section 4.1. Afterwards, in Section 4.2, we explain our data collection approach to collect quantitative and qualitative evidence to answer our RQs. This section includes information about the reasoning behind our collected data, detailed description of our approaches, as well as information about our implementations for the data collection. Additionally, in Section 4.3, we explain the measures that we took to analyze our collected data.

4.1. Overview

Figure 4.1 gives an overview of our study. There are two main RQs that we want to answer with our study presented in this thesis. Within RQ 1, we want to analyze the distribution of unit and integration tests in open-source projects to evaluate if the shift from more unit/-less integration tests to less unit/more integration tests is current and visible in the data. In RQ 2, we analyze the differences between unit and integration tests to evaluate if the aforementioned shift is problematic.

For answering RQ 1 we need to classify the tests of releases from projects and analyze them with respect to our RQ. This RQ is analyzed quantitatively via a case study. How we have chosen our project sample is reported in Section 4.2.2. The results for RQ 1 are presented in Section 5.

We divide the answer of RQ 2 into two different parts: a quantitative and qualitative evaluation of the differences between unit and integration tests. The quantitative analysis answers RQ 2.1-2.3 by extending the case study used in RQ 1. This kind of analysis was not possible for all differences, as information about, e.g., the test costs are not available for open-source projects. Therefore, we perform a qualitative analysis to answer RQ 2.4-2.6.

The quantitative analysis is done in different steps: 1) we need to extract the defect detection capabilities per defect type to evaluate if unit and integration tests detect different types of defects; 2) we need to extract the defect-locality to evaluate if the source of a defect can be found more easily if it was detected by an unit test; and 3) we need to extract the test execution time to evaluate if unit tests are really faster in terms of their execution time in contrast to integration tests. The results of our first RQ help with the aforementioned extractions and the following analysis, as we can reuse the test classification into unit and integration tests. Afterwards, we need to analyze the extracted features to evaluate if we

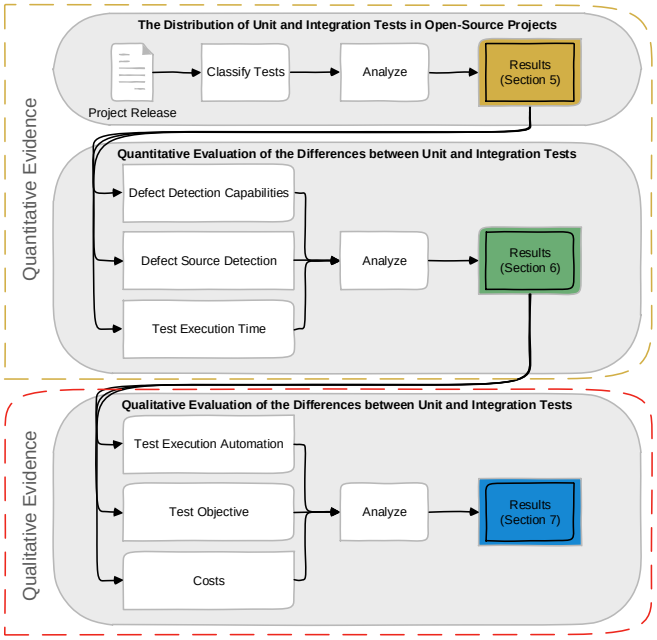


Figure 4.1.: Overview of our study. The yellow dashed line includes the parts of our study that provides us with quantitative evidence, while the red dashed line includes the parts that gives us qualitative evidence on our RQs.

find differences between unit and integration tests with respect to them. The results for the RQ 2.1-2.3 are presented in Section 6.

The qualitative analysis is done by evaluating the current scientific literature, as well as other internet resources to gain an understanding of the scientific and practical view on the difference at hand. Therefore, we analyze scientific literature, developer comments, and the current industrial landscape to evaluate 1) how unit and integration tests are/can be executed automatically; 2) the differences in the test objective for unit and integration tests; 3) how the costs of testing on unit and integration level differ from each other. The results and the knowledge that we gained from our quantitative analysis, helps us with this investigation. In Section 7 we present the results for the RQ 2.4-2.6.

4.2. Data Collection

In the following sections, we give an in-depth explanation of our data collection approach. First, we give a rough overview of our quantitative data collection approach in Section 4.2.1. Then, in Section 4.2.2, we describe our applied inclusion and exclusion criteria to filter for fitting study subjects (i.e., software projects). Afterwards, we explain each step of our data

Difference	Metric
Execution time (D1)	Execution time
Different defects detected (D2)	Defect detection capabilities separated by defect type
Defect source detection (D3)	Defect-locality

Table 4.1.: Differences between unit and integration tests together with the test-specific metric that was chosen to evaluate the differences.

collection that is mentioned in the overview section in-depth, give a reasoning for the approach, and discuss the alternatives. In Section 4.2.3, we explain how and why we extract project meta-data from our study subjects. The extraction of the test level (i.e., the classification of tests into unit and integration tests) is explained in Section 4.2.4. In Section 4.2.5, we describe why and how we extract the Test Lines of Code (TestLOC) and Production Lines of Code (pLOC) of tests. Additionally, in Section 4.2.6, we explain our approach to assess the defect detection capabilities of tests. Afterwards, in Section 4.2.7, we explain how we have classified the defects that were integrated via mutation testing into different defect classes. The extraction of the defect-locality of tests is explained in Section 4.2.8. Then, in Section 4.2.9, we explain our approach to measure the execution time of tests. In Section 4.2.10 we describe the frameworks that we designed, which implement the quantitative data collection approaches described in this section. Finally, in Section 4.2.11, we describe our approach to collect qualitative evidence for answering our RQs.

4.2.1. Overview of our Quantitative Data Collection

Before we can start with the collection of the data, we first need to map the differences between unit and integration tests, that we have extracted from the standard literature, to proxy metrics that we can measure on real open-source projects. Table 4.1 gives an overview of the differences between unit and integration tests that were mentioned in the literature (Section 1.1) together with test-specific metrics which we have chosen as representatives for the differences that we want to evaluate.

We found statements about the execution time in the literature. It is stated, that unit tests are faster than integration tests in terms of their execution time. Hence, we measure the execution time of tests in a standardized environment to minimize influences on the execution time.

We evaluate, if unit and integration tests detect different types of defects by analyzing the defect detection capabilities of these test types. We collect the defect detection capabilities of each test by using mutation testing (Section 2.2). Nevertheless, we do not only want to evaluate the defect detection capabilities, but also what types of defects are detected by

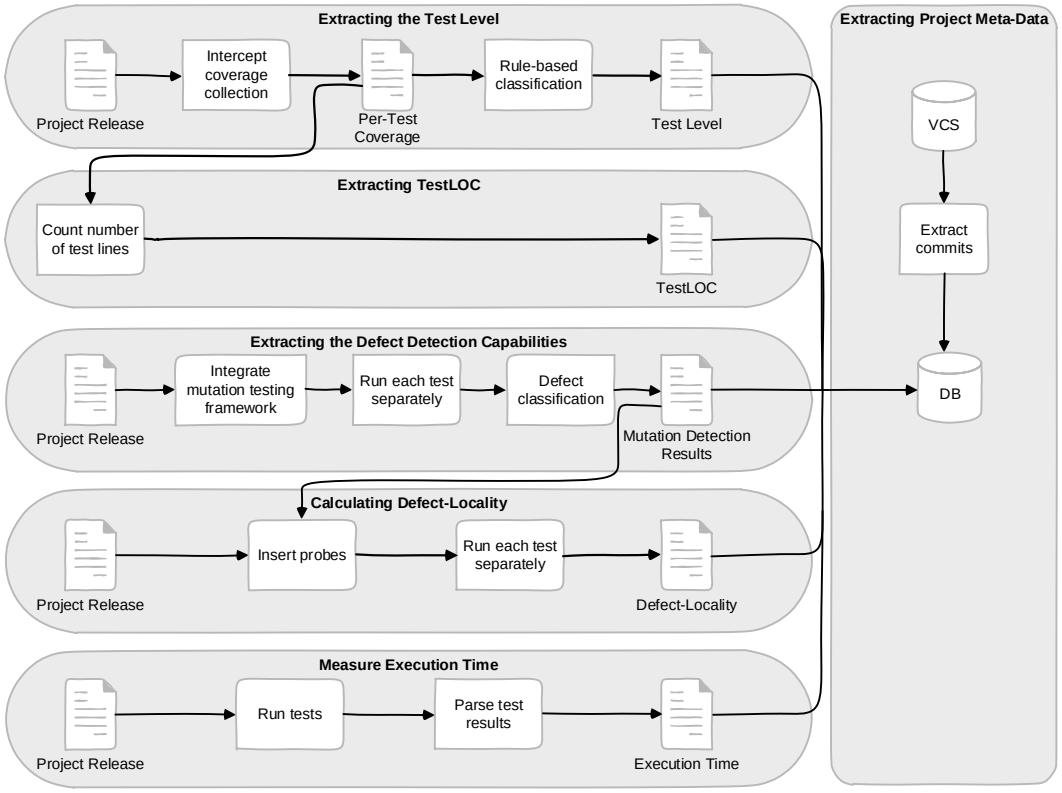


Figure 4.2.: Overview of our data collection.

each test type. Hence, we also need to classify each integrated defect into several defect categories. This is accomplished by adopting a schema by Zhao et al. [141] (Section 4.2.7).

The literature states that the source of the defect can be easier detected if a unit test failed. We designed a new metric for the evaluation of this difference. This metric is called defect-locality and describes the number of methods a developer must debug to find the defect (Section 4.2.8). Basically, it describes the depth of the call stack that a developer must investigate if he wants to debug a found defect.

Figure 4.2 gives an overview of our data collection procedures. Besides the different test specific metrics that must be collected to answer our RQs (Section 4.1), we need additional data to accomplish our goal, e.g., meta-data about the projects and the TestLOC. All test-specific metrics that we are extracting, including the test classification, are test-level metrics. Hence, we follow a rather fine-grained approach instead of calculating the metrics, or the test level, for each test suite as a whole.

As first step, the project meta-data is collected for each project that we selected. The meta-data includes data about commits extracted from the VCS of the project. The mined data is then stored into a database.

The test level classification is collected in the following way. The project release that should be analyzed is checked out. Afterwards, we intercept the coverage collection of the tests of the project to generate a per-test coverage instead of a test suite coverage. This per-test coverage is used in our rule-based classification schema to assign a test level to each test of the project. In the end, the test levels are stored into the database and interconnected with the previously mined data for better accessibility.

For the collection of the TestLOC and pLOC, we reuse the per-test coverage generated during our test level classification. We parse the per-test coverage for each test to count the number of test and production lines of code that were executed. Afterwards, the results are stored into the database and interconnected with the previously mined data.

As mentioned in Section 4.1, we apply mutation testing to collect the defect detection capabilities of tests. For the collection of the mutation detection results, we first checkout the project release. Afterwards, we insert a mutation testing framework into the build process of the project. The framework takes care of the generation of the mutants, as well as the evaluation (i.e., if a test killed a mutant or not). After the mutation testing framework is integrated, we let it run for each test separately, as we want to assess the mutation detection results of each test and not the whole test suite. All mutants that are integrated into the code and which are challenged against the tests are classified. The mutation detection results consist of: 1) which tests were challenged against which kind of mutant; 2) the result of this challenge (i.e., if the test killed the mutant, if the mutant survived, or if the mutant was not covered by the test at all); 3) the classification of each integrated mutant into different defect types. In the end, we store the mutation detection results into the database and interconnect them with previously mined data.

The defect-locality is calculated in the following way. As a fist step, we check out the project release. Afterwards, we insert probes into the source code. These probes are inserted at locations where defects are/were located. These locations are extracted from the mutation detection results. Hence, we extract the location of integrated defects from the mutation detection results and place a probe at this position. We reuse the mutation detection results here, as we know for each test if the test detected the defect that was placed at a certain location. Afterwards, we run each test separately on this instrumented source code. If the test covers the probe, we store the defect-locality for this defect (that is represented by the probe) and the test that ran in our result. In the end, the defect-locality for each test (and each covered defect) is stored into the database and interconnected with previously mined data.

For measuring the execution time we decided for a rather simple approach. First, we checkout the current project release that we want to collect data from. Afterwards, we run its tests to generate the test results. These results are then parsed to extract the execution time. The execution time for each test is then stored into the database and interconnected with previously mined data.

In the following sections, we describe each data collection step in more detail. We set them into the context of the work and state the reasons behind the design of the approach

and its possible alternatives.

4.2.2. Subject Selection

Before we can execute our study, we need to define the study subjects, i.e., software projects. While it would be favorable to apply our approach to as many projects as possible, this would not be feasible, as our approach (especially the data collection, see Section 4.2.1) is partially supported by manual input and we use mutation testing, which is known to be computational expensive [40]. Hence, we need to select a sample of projects for which we can execute our study in a feasible amount of time.

We defined several inclusion criteria for selecting our study subjects:

1. **Projects must be a library or framework.** Library and frameworks are (normally) not executed by themselves, but included and used in other programs. Hence, system tests are less likely to occur. Basically, system tests detect problems in the architecture or design of the system by testing the whole system (e.g., by executing the *main* method and assessing each step of the program) [1]. Libraries are less likely to have system tests, as they offer several entry points into the program through which its functionality can be used (e.g., you can create different distributions or do statistics using the *commons-math* projects, but both of these function have a different entry point into the library). Frameworks, on the other hand, potentially have a single entry point. This is especially true for parser like *jsoup*, *google-json*, or *fastjson*. But, those parsers are used differently than a Java *application*. In a Java application the user gives input to the program (e.g., via the command line), the program processes the input and generates an output. Here, system tests are more likely, as the whole architecture and process flow should be tested from the input through the computational steps taken till the output. But parsers (and frameworks in general) are not used the same way as Java applications. While you call the main entry point for parser, e.g., by giving an HTML document that should be parsed to the method, you do not generate an output and end the program. Instead, you call several functions on the parsed input. Hence, it is more likely that an integration test is written for such a parser, which tests the parsing of the input together with the desired functionality (i.e., testing if the communication between the main class and the desired functionality works correctly). By focusing on libraries and frameworks, we are reducing the risk of miss-classifying a system test as integration test.

2. **Projects must have a minimum of 1000 commits and are at least 2 years old.** We only want to include mature projects in our study, as non-mature projects often do not follow a systematic development process. While mature projects (e.g., projects from the Apache Foundation [169]), have a systematic process and make use of modern software engineering tools like a VCS from which we can use the data later on.

3. **Projects must use Java (6, 7, 8) or Python (2.7 or 3.x) as primary programming/scripting language.** Within our case study, we focus on projects that use Java or Python as programming/scripting language and other languages are out of scope of this thesis. We decided for Java, because it is the most popular programming language according to the TIOBE index[3] [170] and provides a good library and framework support. Python is used in addition, as we wanted to raise the validity of our study by including a scripting language. Furthermore, Python currently gains more and more popularity and is currently the most popular scripting language according to the TIOBE index [170].

4. **Java projects must use Maven [171] as build system, and JUnit [172] or TestNG [173] as test driver.** This is an inclusion criteria that only affects projects that use Java as primary programming language. This is a limitation of our current tooling infrastructure.

5. **Python projects must use unittest [174] or pytest [175] as test driver.** This is an inclusion criteria that only affects projects that use Python as primary programming language. This is a limitation of our current tooling infrastructure.

Moreover, one exclusion criterion is defined:

1. **Projects should not be focused on the Android platform alone[4].** Within our study, we want to focus on pure Java projects. While Android projects also use Java as programming language, there are several differences, especially in respect to the testing of Android applications [176].

After fixing our inclusion and exclusion criteria, we used them on two different data sources. First, the list of Borges et al. [177, 178], which classified the most popular 5000 GitHub repositories (language-independent) into six different categories (i.e., application software, system software, web libraries and frameworks, non-web libraries and frameworks, software tools, and documentation). Second, we used a list of the most popular Java libraries created by the MVN Repository [179][5]. Both of these lists provided us with a pre-classification for projects, so that we can sample from the libraries and frameworks. Overall, 44 Java projects fitted our selection criteria from which we randomly selected 17 projects for our study. Additionally, from the 55 fitting Python projects, we randomly selected 10 projects for our study. Unfortunately, we could not select more projects because of the time constraints of this thesis.

Table 4.2 gives an overview of the selected projects together with some characteristics. It highlights the name of the project, the release that was used in our study, the overall number

[3] As available on the time of our analysis (January 2018)

[4] For example, Android widgets like `https://github.com/2dxgujun/AndroidTagGroup` or `https://github.com/zcweng/ToggleButton`.

[5] As available on the time we performed our analysis (January 2018).

of commits (till the stated release), the number of files (i.e., *.java* files for Java projects and *.py* files for Python projects), and the number of tests that are executed by the build script of the release. All of our selected projects can be found on GitHub [180]. The number of programming files together with the number of commits and tests highlight, that we only included non-trivial projects in our study.

We included the latest minor release of the projects that we sampled as analysis subject, if it was available. If this release was not available, we took the next possible one (e.g., release 1.11.1 of jsoup was taken, as release 1.10.0 was not available). The minor release was determined, by looking at the available releases on the GitHub pages of the projects. Most of the projects used the semantic versioning release naming schema [181]. Hence, all releases have the form of XX.YY.ZZ, where the number at the position XX was raised if a new major version was published, YY for a new minor version, and ZZ for a bugfix version. For projects that use only two numbers (i.e., XX.YY) we considered the XX position as major and YY as minor versions. For two of the projects that we sampled (i.e., *commons-beanutils* and *fastjson*) the latest minor release was longer than four years ago. We took the most recent release to prevent compatibility issues with our infrastructure.

4.2.3. Extracting Project Meta-Data

The project meta-data that we need for our study comprises of data from the VCS of the projects. The collection of this meta-data allows a better accessibility of our results due to the interconnection between the meta-data itself and the data collection results. Finally, it eases the analysis of the data by working as the ”glue“ between the different data collection results.

Nevertheless, mining software repositories (especially VCS and Issue Tracking System (ITS)) is often hard, as there exist different VCS and ITS that are used nowadays [209, 210]. These different systems also store different data or they store data differently. For example, Apache Bugzilla [211] has other data fields for which data is stored than Jira [212]. Hence, there is a need for a model that is able to reflect the contents of different VCS and ITS.

During our work, we designed such a model. The model is shown in Figure 4.3. The advantage of the model shown in the figure is its general applicability, as it can be used to combine data from different VCSs. The yellow box depicts data that is extracted from the VCS. In addition, there is the project entity which is the starting point for our meta-data extraction. Therefore, it is not directly extracted from a software repository, but is used to link different software repositories of a project together, e.g., other repositories like ITSs can be linked to the project.

We designed and developed different programs that implement the model shown in Figure 4.3. They extract the data from the corresponding software repository, transform the data so that it fits the model, and store the model into the database. More information on the implementation can be found in Section 4.2.10.1.

Project	Release	#Commits	#Files	#Test Cases
commons-beanutils [182]	1.9.3	1128	257	1197
commons-codec [183]	1.11	1677	124	874
commons-collections [184]	4.1	2852	525	6637
commons-io [185]	2.5	1868	227	1150
commons-lang [186]	3.7	5106	331	4064
commons-math [187]	3.6	5819	1616	6488
druid [188]	1.1.0	5178	3314	4132
fastjson [189]	1.2.41	2579	2496	4176
gson [190]	2.8.0	1306	193	1014
guice [191]	4.1	1513	548	701
HikariCP [192]	2.7.0	2493	83	120
jackson-core [193]	2.9.0	1323	228	774
jfreechart [194]	1.5.0	3622	990	2175
joda-time [195]	2.9	1913	329	4176
jsoup [196]	1.11.1	1106	105	593
mybatis-3 [197]	3.4.0	1816	1004	1053
zxing [198]	3.3.0	3282	485	401
ChatterBot [199]	0.8.0	1411	176	299
csvkit [200]	1.0.0	1279	47	177
dpark [201]	0.4.0	1057	59	57
mrjob [202]	v0.6.0	6437	311	1862
networkx [203]	networkx-1.11	4201	432	2723
pyramid [204]	1.9	10357	462	2626
python-telegram-bot [205]	v10.1.0	1791	250	638
rq [206]	v0.10.0	1302	46	207
schematics [207]	v2.0.0	1295	2099	348
scrapy [208]	1.5.0	6568	357	1641

Table 4.2.: Selected projects with their characteristics. In the number of files only *.java* files are included for Java projects and *.py* files for Python files. The dashed line separates the Java projects (upper part) from the Python projects (lower part).

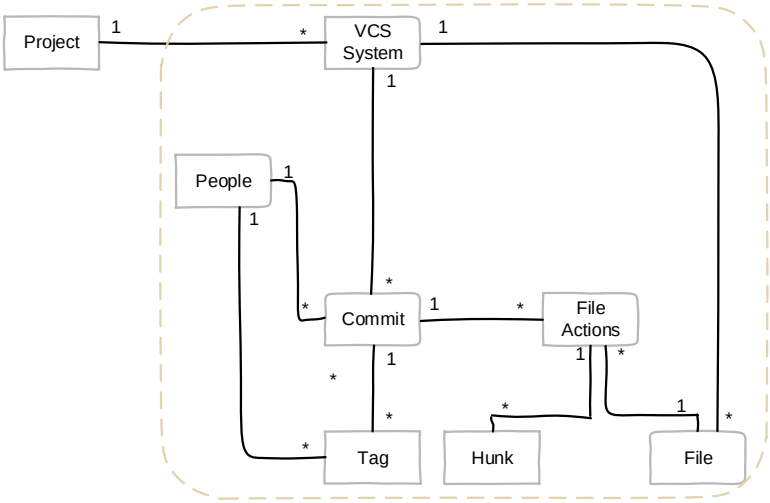


Figure 4.3.: Model that describes the project meta-data that is collected. The yellow box depicts data that is extracted from the VCS.

4.2.4. Extracting the Test Level

We developed three different rule sets to classify tests into unit and integration tests. While the first two rule sets (ISTQB and IEEE) have the definitions of the ISTQB and IEEE as basis (Section 2.1.2), the third rule set (DEV) represents the developer classification of a test and is based on popular coding conventions [119, 120].

Regardless of the rule set that is used, the input is the same for all three: coverage data that is recorded for all tests separately. Hence, the coverage recording includes a list of executed tests together with their covered units. Beforehand, we filter out every test class that might be in the coverage data, as we want to base our test level assignment only on the covered production classes. We look at the path of each covered unit and if this path contains the word “test”, we assume it is a test class and a production class otherwise. We exclude units that have “test” or “validate” as part of their name. A unit counts as covered, if at least one method of the unit was covered by the test. In addition, we only record the coverage of units that are *within* the project. Hence, if a test or another unit calls functions from a framework that are outside the scope of the project, we do not cover its execution. The reasoning behind this is that we want to focus on the project that we are analyzing and what is really tested inside this project. Besides these filtering of the tested units, we also need to filter out tests that can not be classified correctly. Hence, we are filtering out:

- tests that are skipped by the test execution framework.
- tests that are empty.

- tests that only test constants but no functions.
- tests that test other projects but not the project at hand (e.g., tests that ensure the correct working of java.io).
- tests that test the test setup itself (e.g., testing if the database is correctly running, but not executing any production code).
- tests where an exception is directly thrown after the first method call (if this is the case, no coverage gets collected and we can not classify the test).

The resulting tests and their tested units are then used to assign a test level to each test separately. Table 4.3 shows the rules that we designed for each rule set. A test is a unit test in respect to the IEEE rule set if it only covers units from within one package. This represents the IEEE definition, where a unit test is a test that tests several units that are logically connected. Within Java, related units are put into one package, as the official Java documentation states [213]. Same goes for Python projects [214]. However, if the test covers units from more than one package, it is classified as integration test. The ISTQB definition is stricter than the IEEE definition. A test is a unit test in respect to the ISTQB rule set if it covers only one unit. If it covers more than one, the test is classified as integration test. We developed several rules to represent the developer classification of a test based on coding conventions. Hence, if a test is matching the name of a unit (e.g. if we have a unit called *Fnatic* and a test called *FnaticTest*) or the path to the test has the term “unit” in it (e.g., *src/test/java/de/ugoe/unit/FnaticTest.java*) it is classified as a unit test. If there is no unit matching the name of the test or the path to the test has the term ”integration“ or ”IT“ in it (e.g. *src/test/java/de/ugoe/FnaticITTest.java*) it is classified as integration test. It is important to mention that we do not classify the *intent* of a test, but its actual type according to the definitions. Hence, we do not evaluate if the test *should* be a unit test, but if it *is* a unit (or integration) test.

Figure 4.4 gives some examples of our classification schema. The figures depict schematic call graphs. The first figure in Figure 4.4 shows that a test only calls one unit from within one package. If we apply the above explained classification schema the test gets classified as a unit test for the IEEE and ISTQB definitions. This is different for the second figure in Figure 4.4. Here, two different units are called from test t_1 which both reside in one package. Hence, the test gets classified as an unit test for the IEEE definition, but as an integration test for the ISTQB definition. The third figure in Figure 4.4 depicts a test that is classified as an integration test for both definitions, because the test calls two different units from two different packages. A more complex example is given in the forth figure in Figure 4.4. While the test only calls one unit directly, this unit calls other units from within other packages. Hence, the coverage data for test t_1 would include all four units from two different packages. Therefore, this test gets classified as an integration test for both definitions.

	Unit Test Classification Rules	Integration Test Classification Rules
IEEE	• Only covers units from within one package	• Covers units from more than one package
ISTQB	• Only covers one unit	• Covers more than one unit
DEV	• A unit matching the name of the test • Path to the test has "unit" in it	• There exist no unit matching the name of the test • The path to the test has "integration" or "IT" in it

Table 4.3.: Rule sets for our test level classification.

To foster the understanding of our classification approach, we included two different real world tests from the *commons-io* project. Listing 4.1 shows an IEEE/ISTQB unit test. This test asserts the correct workings of the *getPrefix* function of the *FilenameUtils* class. More precisely, this tests checks if *getPrefix* returns the correct string if the input string contains null bytes. As this test only calls one unit (i.e., the *FilenameUtils* class) and the class itself does not call other units, this tests gets classified as unit test for both definitions.

```
1  [...]
2  package org.apache.commons.io;
3  [...]
4
5  @Test
6  public void testGetPrefix_with_nullbyte() {
7    try {
8      assertEquals("~user\\", FilenameUtils.getPrefix("~u\
           u0000ser\\a\\b\\c.txt"));
9    } catch (IllegalArgumentException ignore) {
10   }
11 }
```

Listing 4.1: Example of an unit test from the *commons-io* project [185].

On the other hand, Listing 4.2 depicts a test, which is classified as integration test for both definitions. Here, the test checks if the *ByteArrayOutputStream* class of *commons-io* works as intended if it is used within the *copy* function of the *CopyUtils* class. As the *CopyUtils* class is from the *org.apache-commons.io* package and the *ByteArrayOutputStream* class from the *org.apache.commons.io.output* package, this test gets classified as an integration test (according to both definitions).

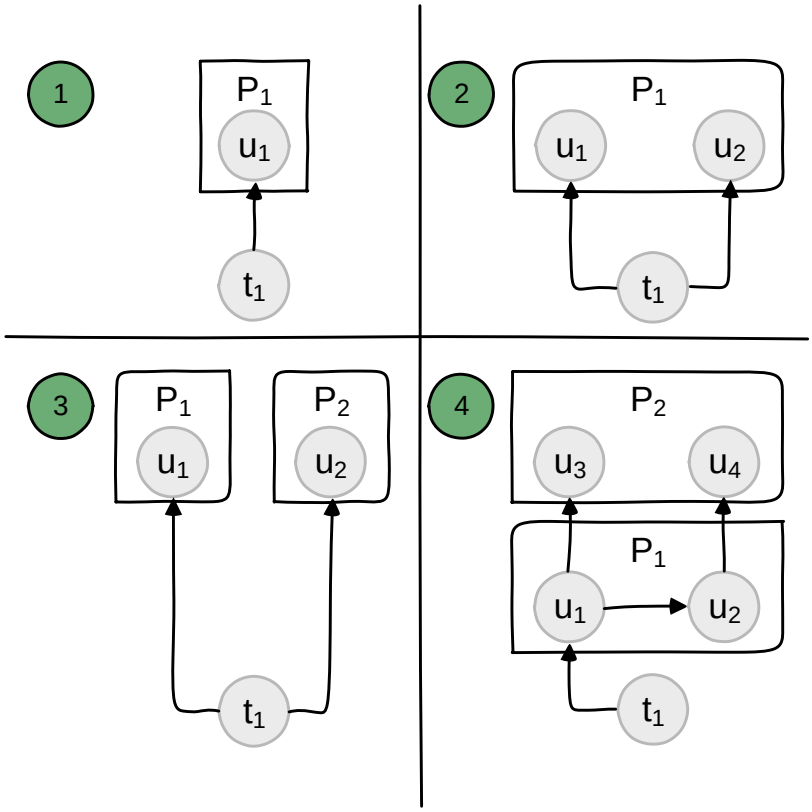


Figure 4.4.: Different example call graphs. t_1 depicts a test, u_x depict different units and P_x different packages. 1) IEEE/ISTQB unit test; 2) IEEE unit test/ISTQB integration test; 3) IEEE/ISTQB integration test; 4) IEEE/ISTQB integration test.

```
1  [...]
2  package org.apache.commons.io;
3  [...]
4  import org.apache.commons.io.output.ByteArrayOutputStream;
5  [...]
6
7  @Test
8  public void copy_byteArrayToOutputStream() throws Exception
       {
9    final ByteArrayOutputStream baout = new
         ByteArrayOutputStream();
10   final OutputStream out = new
         YellOnFlushAndCloseOutputStream(baout, false, true);
11
12   CopyUtils.copy(inData, out);
13
14   assertEquals("Sizes differ", inData.length, baout.size());
15   assertTrue("Content differs", Arrays.equals(inData, baout.
         toByteArray()));
16 }
```

Listing 4.2: Example of an integration test from the *commons-io* project [185].

Besides the classification via per-test coverage data, we also evaluated other classification methods. Unfortunately, none of them could be reused. Orellana et al. [118] proposed to classify tests based on with which Maven plugin they are executed (i.e., Maven Sure-Fire [119] or Maven FailSafe [120]) (Section 3.1). The problem with this approach is its applicability, as it can only be applied to Java projects which use Maven and have configured the use of both of the mentioned Maven plugins. Thus, the number of projects that can be analyzed via this approach is rather limited. For example, none of the projects that we have used in our study (Section 4.2.2) make use of the Maven FailSafe plugin, while some of the tests of the projects are indeed integration tests. In addition, the approach by Orellana et al. [118] classifies the whole test suite instead of a single test. A deeper analysis based on tests and not test suites would not be possible.

Another possible classification method is the differentiation of tests based on their assertions. If a test only asserts one unit, it is classified as a unit test and as an integration test otherwise. But, during the evaluation of this approach we found that developers sometimes use only one assert, even for integration tests. They call several units during the test, but only assert one unit, e.g., to check if the communication from one to another unit was working (e.g., by asserting if a flag in the receiver unit was set).

In contrast to other approaches, our approach is fine-grained, as we classify each test separately instead of each test suite only. Therefore, our level of detail is higher and better reflects the development reality. Furthermore, our approach is dynamic in contrast to other approaches that make use of static analysis of the tests and production units. The downside is that we need to execute the tests before we can analyze them to collect the coverage data, which can be difficult, as Tufano et al. [215] highlight. Nevertheless, a dynamic analysis has the advantage that we get more reliable results, as a static analysis approach could not handle techniques like dependency injection [122], reflection [122], other dynamic structures (e.g., functions that can be given as parameters of other functions in Python), or the usage of mocking frameworks (e.g., [216, 217]), which are commonly used nowadays.

4.2.5. Extracting TestLOC and pLOC

We need to acquire the number of TestLOC and pLOC of each test as it is later on used for the normalization of results and it is a more detailed metric than the number of tests. In our approach, this is done by reusing the per-test coverage data (Section 4.2.4). The number of covered lines of each test case is summed up, while we differentiate between pLOC and TestLOC. This differentiation is done based on the path in which the covered class resides. If the path contains the word “test” we know that the covered class is a test class (e.g., for Maven projects all test classes reside in the “src/test” folder). Hence, all covered lines for this class count to the covered test lines and to the production lines otherwise.

In contrast to just counting Lines of Code (LOC) or Logical Lines of Code (LLOC) this approach has the advantage that we consider all executed test code instead of just the lines of the test case itself. Our results could be biased otherwise. For example, the size of a test

that calls many other test methods (e.g., to bring several units into a testable state) would not be correctly represented by LOC or LLOC, as these other calls would not count to the LOC of the test method itself. Thus, we decided to make use of our recorded coverage data and use the number of TestLOC and pLOC.

4.2.6. Extracting the Defect Detection Capabilities via Mutation Testing

The assessment of the defect detection capabilities of a set of tests is done by checking how many defects the tests can find. However, a controlled environment where as many outside influences as possible are excluded, need to be created [124]. Furthermore, we need an approach that is able to also cover the defects that are found by the developers before they are committing their changes to the VCS. As explained in Section 4.1, we decided on mutation testing (Section 2.2).

Mutation testing is an approach that is often used in research to evaluate the defect detection capabilities of tests. Nevertheless, as Papadakis et al. [40] report, there are several pitfalls one can encounter when using mutation testing. In our approach, we reused the knowledge of Papadakis et al. [40] and followed their best practices on using mutation testing in controlled experiments. As advised by Papadakis et al. [40] we report on each decision regarding our application of mutation testing.

Mutation Testing Framework: The choice of the correct mutation testing framework is crucial for the validity of the analysis. Mutation testing frameworks are implementing the idea of mutation testing for a specific language (or languages). They provide different mutant operators that can be applied to the source code and they ease the execution, e.g., by making mutation testing available through build management systems. There are several mutation testing frameworks for different languages available.

Table 4.4 shows a list of mutation testing frameworks for Java and Python. We do not list frameworks for other programming languages here, as it is out of scope for this thesis. As Table 4.4 highlights there are several mutation testing frameworks for Java available, while there exist only two frameworks for Python. We evaluated the listed frameworks for their applicability in our approach to gather the mutation detection capabilities of tests. We had the following requirements for a mutation testing framework.

- It must be open-source, so that we can evaluate and comprehend how the mutation testing is implemented.

- It must support prominent test runners for Java (i.e., JUnit4 [172]) and Python (i.e., nosetests [218] and pytest [175]). Otherwise, we could not integrate the mutation testing framework into our selected projects.

- It is actively supported and developed. This aspect is important, as the framework might have bugs or problems with newer technologies or program versions, e.g., newer Java versions.
- The mutant selection technique that is used by the framework should be coverage-based. Only mutants that are covered by the test should be injected. This reduces the execution time.
- The framework should show a good defect-revelation ability, which is an indicator of its suitability for our experiments [219].
- It must work with different Java (i.e., Java 6, Java 7, Java 8) and Python versions (i.e., Python 2.7 and Python 3.5).

For the Java programming language, we selected PIT [67] as mutation testing framework as it fulfills all of our requirements. There were also other mutation testing frameworks for Java that fulfilled our requirements (e.g., PIT_{RV} or Major), but a recent study of Kintis et al. [219] showed that PIT has a better defect-revelation ability than, e.g., Major which is crucial for our study. Nevertheless, PIT_{RV} was able to reveal more defects (115:122), but the number of *equivalent* mutants is substantially higher for PIT_{RV} in contrast to PIT. In fact, Kintis et al. [219] discovered in their experiment that PIT_{RV} generates 88.74% more equivalent mutants than PIT. We decided to use PIT instead of PIT_{RV}, as equivalent mutants are a substantial threat to the validity of our study. We contributed to the development of PIT by creating a pull request to improve PIT. This pull request included a feature that allows PIT users to filter the tests of a test suite that should be challenged against the generated mutants. The pull request got accepted and integrated into the version 1.3.2 of PIT, which we also used for our study.

Unfortunately, both frameworks that we evaluated for Python (i.e., MutPy and Cosmic Ray) did not fulfill our requirements. Cosmic Ray contained bugs and was not working for the projects that we tested it on. The reason for this is that Cosmic Ray is not a finished product, as we can see in the documentation which states: ”At this time Cosmic Ray is young and incomplete. It doesn't support all of the mutants it should, its output format is crude, it only supports some forms of test discovery, it may fall over on exotic modules...the list goes on and on.“ [220]. Furthermore, it does not support Python 2.7. MutPy does not support Python 2.7 and it also does not support pytest as test runner, which is used by several projects that we selected.

Name & Ref	Year	Application	Description
Jester [221]	2001	Java	supports source-code-level (src-level) mutant generation
MuJava [222, 223, 66]	2004	Java	implements src-level mutant generation and supports method-level and Object-Oriented (OO) mutant operators
ByteME [224]	2006	Java	implements bytecode-level mutant generation and supports method-level and OO mutant operators
Jumble [225]	2007	Java	implements bytecode-level mutant generation and supports method-level mutant operators
Javalanche [226]	2009	Java	implements bytecode-level mutant generation and supports method-level mutant operators and mutant classification based on mutants' impact
Not Named [227]	2009	Java	supports mutant operators that follow the fault classification of Durães and Madeira [228]
PIT [67]	2010	Java	implements bytecode-level mutant generation and supports method-level mutant operators
MutMut [229]	2010	Java	supports concurrency-related mutant operators
Judy [230]	2010	Java	implements src-level mutant generation and supports method-level and OO mutant operators
Bacterio [231]	2010	Java	supports method-level mutant operators for system-level testing using flexible weak mutation
Major [232]	2011	Java	supports method-level mutant operators
Paraμ [233]	2011	Java	supports OO and concurrency-related mutant operators and higher order mutation
Not Named [234]	2011	Java	supports method-level mutant operators based on the selective mutation approach and higher order mutation
Comutation [235]	2013	Java	supports concurrency-related mutant operators [236]
HOMAJ [237]	2014	Java	supports higher order mutation
PIT_{RV} [67, 238]	2016	Java	has PIT as basis, but introduces more mutation operators

Table 4.4.: Mutation testing tools for Java and Python. Based on [40].

Name & Ref	Year	Application	Description
LittleDarwin [239]	2017	Java	supports method-level mutation operators, higher order mutation, mutant sampling and disjoint/subsuming mutant analysis
Not Named [51]	2017	Java	Implements security-aware mutation operators
MutPy [240]	2014	Python	implements traditional and python-specific mutation operators
Cosmic Ray [241]	2017	Python	implements traditional and python-specific mutation operators

Table 4.4.: Mutation testing tools for Java and Python. Based on [40]. (Continued)

Mutant Redundancy: Recent research showed that redundant mutants are a substantial threat to the validity of empirical research. Andrews et al. [137] state, that in order to have a representative relation between mutants and real faults it might be important to filter out trivial mutants. Visser [242] suggested to identify mutants that are hard to kill by controlling for the reachability of mutants. Recent empirical studies showed that the subsumed mutant threat really endangers the validity of studies that make use of mutation testing. Kurtz et al. [243] did a replication of earlier studies on selective mutants and found that the approaches performed well when redundant mutants are included, but poorly when they are discarded.

The quality of the employed mutants is a major concern when mutation testing is used in a study. If the mutants are rather trivial (i.e., they are found by nearly all tests), we are only measuring the ability of test suites to cover the code of the project instead of their defect-revelation ability. This kind of threat is called the "subsumed mutant threat" [42]. This problem is especially important if we generate a large number of mutants, like we do in our study. Recently, Papadakis et al. [42] estimated that more than 60% of scientific conclusions (for arbitrary experiments) that were made on the basis of mutation testing are endangered, because of the subsumed mutant threat.

Researchers proposed different solutions to tackle and ease the subsumed mutant threat. Kaminski et al. [244, 245] and Just et al. [246] proposed to remove logical and relational mutation operators to ease the threat. Ammann et al. [247] adopted the notion of minimal mutants, which is the smallest possible set of mutants, while Kurtz et al. [248] suggested mutant subsumption graphs to select the minimal set of mutants. Other solutions include using symbolic execution to approximate subsuming mutants (e.g., [249]) or using compiler optimizations to remove duplicated mutants[6] (e.g., [43, 71]).

[6]Duplicated mutants are also redundant mutants.

Kintis et al. [250] proposed another solution: they suggested to create a set of disjoint mutants. This disjoint mutant set is a representative subset of all mutants. A representative subset is defined as follows: "Consider that we have a set of N mutants, a representative subset, say D, means that any test suite that kills this subset of mutants also kills the N mutants. No redundancy between the mutants of D means that it is not safe to remove any mutant from this set because in this case we fail to kill all the N mutants" [40]. The difference between the disjoint mutant set and the minimal mutant set proposed by Ammann et al [247] is that the disjoint mutant set might not be minimal, but it does not have any redundancies. Unfortunately, computing the true disjoint mutant set is impossible and it can only be approximated. Algorithm 4.1 computes a dynamic approximation of the disjoint mutant set in a greedy manner. First, live and duplicate mutants are removed (lines 2 and 3). Afterwards, the subsuming mutant with the largest number of live mutants is selected (lines 9-16). This mutant is found by looking at test cases: It "is the mutant that is killed by test cases, which manage to collaterally kill the highest number of other mutants" [40]. The found mutant is then added to the set of disjoint mutants (*D*, line 18). Then, the joint mutants are removed from the set of mutants (*S*, line 19). The aforementioned process is repeated until the set of mutants (*S*) is empty. In the end, the set of disjoint mutants is returned.

As mutant redundancy might have a large impact on the validity of our study, it is important to care for this kind of threat. Checking all generated mutants by hand to detect redundant mutants is not feasible and error prone for the number of mutants that we generated within our experiments (> 500.000). Furthermore, leveraging an algorithm like JudyDiffOp [70] is still very time-intensive and not feasible for the size of our study. Hence, we decided to reuse Algorithm 4.1, as proposed by Kintis et al. [250] to reduce this threat to our study.

Mutant Selection: The selection of mutation operators is another crucial point for a mutation testing study. There exist a lot of different mutation operators, e.g., operators for specific programming languages (e.g., [47, 251, 252, 253]), operators for specific categories of programming languages (e.g., [254, 255, 256]), operators for specific categories of applications (e.g., [48, 257, 258, 259]), as well as operators for specific categories of bugs (e.g., [52, 53, 260, 261]).

In our approach, we decided to reuse all mutation operators that are provided by our mutation testing framework. The reasoning behind this approach is that we wanted to generate a huge set of mutants to reduce the possibility that we only create low quality mutants (i.e., mutants that do not reflect real software defects). Kintis et al. [219] provide empirical evidence that supports this approach. In addition, by having a great variety of mutation operators we reduce the risk that we only integrate defects of a certain type. We do not only want to integrate defects that can be found by unit testing, but also integrate interface problems that should be found by integration tests. Table 4.5 depicts all mutation operators that

Algorithm 4.1 Algorithm to dynamically approximate the set of disjoint mutants. Based on: [40]

Require: A set S of mutants
Require: A set T of tests
Require: A matrix M of size $|T|x|S|$ such as $M_{ij} = 1$ if test$_i$ kills mutant$_j$
1: $D = \emptyset$
2: $S = S \setminus \{m \in S | \forall i \in 1..|T|, M_{ij} \neq 1\}$ // Remove live mutants
3: $S = S \setminus \{m \in S | \exists m' \in S | \forall i \in 1..|T|, M_{ij(m)} = M_{ij(m')}$ // Remove duplicate mutants
4: **while** $|S| > 0$ **do**
5: maxJoint $= 0$
6: jointMut = null
7: maxMutDisjoint = null
8: // Select the most disjoint mutant
9: **for all** $m \in S$ **do**
10: sub$_m = \{m' \in S | \forall i \in 1..|T|, (M_{ij(m)} = 1) \Rightarrow (M_{ij(m')} = 1)\}$
11: **if** $|\text{sub}_m| > maxJoint$ **then**
12: maxJoint = $|\text{sub}_m|$
13: maxMutDisjoint = m
14: jointMut = sub$_m$
15: **end if**
16: **end for**
17:
18: $D = D \cup \{\text{maxMutDisjoint}\}$ // Add the most disjoint mutant to D
19: $S = S \setminus$ jointMut // Remove the joint mutants from the remaining
20: **end while**
21: **return** The disjoint mutant set D from S

we used in our study to generate mutants together with a short description. For example, the "Argument Propagation" operator replaces calls to non-void methods with one of the arguments and the "Constructor Calls" operator replaces calls to constructors with `null`.

We note, that we reuse several mutation operators that are also used in mutation testing approaches that are focused on integration testing (e.g., [262, 263]). Thus, with the selection of these mutation operators we balance out the generated mutants that should be found by unit and integration testing.

Mutation operator	Description
AP: *Argument Propagation*	$\{(nonVoidMethodCall(...,par),\ par)\}$
BFR: *Boolean False Return*	Replaces primitive and boxed boolean return values with `false`.
BTR: *Boolean True Return*	Replaces primitive and boxed boolean return values with `true`.
CB: *Conditionals Boundary*	$\{(op_1,op_2) \mid (op_1,op_2) \in \{(<,<=),(<=,<),(>,>=),(>=,>)\}\}$
CC: *Constructor Calls*	{(`new` *AClass*(), `null`)}
EOR: *Empty Object Return*	Replaces return values with an empty value for that type. For example, if the return type is a string it returns "", if it is a `List` an empty `List` will be returned.
I: *Increments*	$\{(op_1,op_2) \mid op_1,op_2 \in \{++,--\} \wedge op_1 \neq op_2\}$
IC: *Inline Constant*	$\{(c_1,c_2) \mid (c_1,c_2) \in$ {(1,0), ((`int`) x, `x+1`), (1.0, 0.0), (2.0, 0.0), ((`float`) x, 1.0), (`true`, `false`), (`false`, `true`)}}
IN: *Invert Negatives*	$\{(v,-v)\}$
M: *Math*	$\{(op_1,op_2) \mid (op_1,op_2) \in$ {(+,-), (-,+), (*, /), (/, *), (%, *), (&,|), (^, &), $(<<,>>),(>>,<<),(>>>,<<)$ }}
MV: *Member Variable*	{(`member_var=...`, `member_var=b`) | b ∈ {`false`, 0, '\u0000', 0.0, `null`}}
NR: *Naked Receiver*	Replaces a method call with the receiver for non-void methods where the return type matches the receiver's type.
NC: *Negate Conditionals*	$\{(op_1,op_2) \mid (op_1,op_2) \in \{(==,!=),(!=,==),(<=,>),(>=,<),(<,>=),(>,<=)\}\}$
NVMC: *Non Void Method Calls*	{(nonVoidMethodCall(), c) | c ∈ {`false`, 0, 0.0, '\u0000' , `null`}}
NR: *Null Return*	Replaces return values with null.
PR: *Primitive Return*	Replaces `int`, `short`, `long`, `char`, `float` and `double` return values with 0.
RC: *Remove Conditionals*	Removes or negates a conditional statement to force or prevent the execution of the guarded statements, e.g. {((a *op* b), true) *or* ((`LHS && RHS`), `RHS`)}
RI: *Remove Increments*	$\{(--v,v),(v--,v),(++v,v),(v++,v)\}$
RS: *Remove Switch*	Changes all labels of the `switch` to the default one.

Table 4.5.: Mutation operators of PIT. Based on the table by Kintis et al. [219].

Mutation operator	Description
RV: *Return Values*	{(`return a`, `return b`) | (a,b) ∈ {(`true`, `false`), (`false`, `true`), (0, 1), ((`int`) `x`, 0), ((`long`) `x`, `x+1`), ((`float`) `x`, `-(x+1.0)`), (`NAN`, 0), (`non-null`, `null`), (`null`, `throw` RuntimeException)}}
S: *Switch*	Replaces the `switch`'s labels with the default one and vice versa (only for the first label that differs)
VMC: *Void Method Calls*	$\{(voidMethodCall(), \varnothing)\}$

Table 4.5.: Mutation operators of PIT. Based on the table by Kintis et al. [219]. (Continued)

Test Suite Choice and Test Suite Size: For the scope of our analysis, the test suite choice and size is predetermined. We want to assess the mutation detection capabilities of the tests that are inside the test suites of our selected projects. Therefore, we reuse the test suites that were created by the developers of the projects.

Clean Program Assumption (CPA): Basically, assessing tests via mutation testing can be seen as a simulation that involves two "roles": the faults role (mutants) and the oracle role (original program). Aligning this simulation to the reality, "we can say that developers produce the faulty programs (simulated by the mutants) which they test using a test oracle (simulated by the original program)." [40]. Testers then apply their techniques in the mutant program version to check for unexpected behavior (as defined by the oracle) and report found bugs [40].

Nevertheless, test assessment is practiced differently. It is common to apply test techniques on the original program (not the faulty one) and check their fault-revealing power by executing tests on mutants. While this is less time-consuming than applying the test technique to each faulty program version, it makes an implicit assumption (called "CPA") that the "coverage measurements (or the application of test techniques) on the original program are representative (or very similar) of those on the mutant programs" [40]. The problematic part is, that test suites are assessed on the mutated program version instead of the original one (for which they were created).

The first empirical study that evaluated the CPA was performed by Chekam et al. [86]. The results of this study showed, that we can not rely on the CPA. Chekam et al. [86] highlighted that the CPA has an influence on the outcome of empirical studies, if the CPA is not controlled for.

While we do know of the CPA, we do not need to take it into account for our experiment in this thesis. As we do not compare different testing techniques with each other or rely on coverage that were measured on the clean program.

Multiple Experimental Repetitions: Techniques that make stochastic choices should be assessed by multiple experimental repetitions, as the results can be skewed otherwise [264, 265]. Often, this might not be practical as mutation testing is very expensive in terms of computation time [40]. Papadakis et al. [40] suggest, that researcher should then do their experiments with only few subjects, but many repetitions. There even exist some work that supports this approach [266].

In our study, we generate all mutants separately for each test to create the matrix needed for the calculation of the disjoint mutant set. The calculations for our empirical study are only done once, because we do not have a stochastic choice in it. Hence, a rerun of our whole experiment would result in the same data collected. One exception is the creation of the disjoint mutant set presented above. As Algorithm 4.1 is non-deterministic, we need to apply multiple experimental repetitions. In our case, we repeated the generation of the disjoint mutant set 10 times.

Presentation of the Results: Papadakis et al. [40] emphasize that the granularity level of the empirical results should be clarified. Our results are generated and presented on test level as our collected mutation detection capabilities data contains information about all generated mutants together with which test killed it or failed to do so.

After the scope of our mutation analysis is set, we clarify the execution of our data collection. Basically, we collect mutation detection results (i.e., which mutant was killed or survived) for each test separately. This is done, because Algorithm 4.1 needs (among others) a mutant killing matrix as input. This mutant killing matrix can only be provided, if we execute our mutation testing framework (in our case PIT) for each test separately. This is because PIT directly terminates if a test killed the mutant. As a result, we would only know the first test that killed the mutant, but not if other test would kill it as well. Normally, this would not be problematic as one would only wants to know if a mutant was killed by any test of the test suite. But, for our analysis we need data about which test killed which mutant (i.e., the mutant killing matrix). We isolate the execution of tests by running PIT for every test of the project. Afterwards, the generated mutants are classified into different defect categories. How this is handled is further described in Section 4.2.7. In the end, the output of PIT is parsed and stored into the database.

4.2.7. Defect Classification

As presented in Section 3.3 there are numerous studies that propose a classification of defects. Nevertheless, not all of these approaches are suitable for our purpose. As we are using open-source projects in our study, we can not reuse classification schemes that are

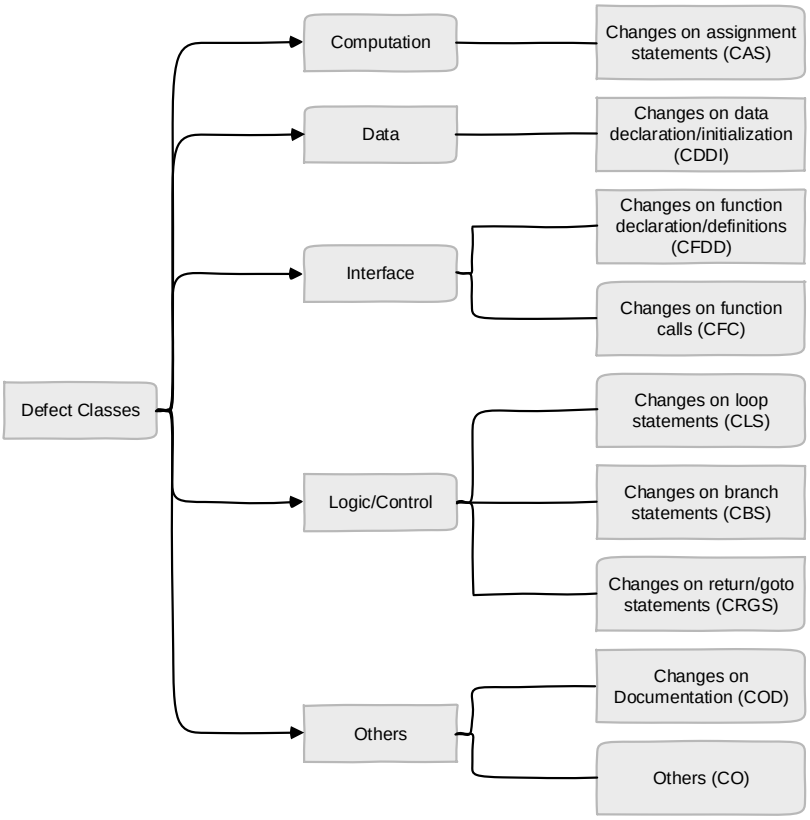


Figure 4.5.: Overview of our defect classification scheme. Figure adopted from [141].

based on external data like, e.g., design documents or specifications [140]. These kind of data are often not available in open-source projects, as they follow a special development process [146]. As we want to classify artificial defects, classification schemes that need additional defect data (e.g., defect reports [142, 143]) are also not suitable. Therefore, we decided to use a classification scheme that only needs the code for the classification. Hence, we adapted the classification scheme of Zhao et al. [141].

Figure 4.5 depicts our defect classification scheme. In comparison to the scheme by Zhao et al. [141] we excluded the CPD category, as pre-processor directives are not available in Java[7]. In addition, we included the "Changes on Documentation (COD)" to be classified as "Others" as well. In contrast to Zhao et al. [141], which followed a fine-grained approach by classifying defects in accordance to the sub-categories, we decided to only collect the top-level class (i.e., computation, data, interface, logic/control, or others) of a defect. The reason behind this is that the data would be too scattered, i.e. many categories with too few

[7]Recall, that we only collect the defect detection capabilities for Java projects due to a missing mutation testing framework for Python.

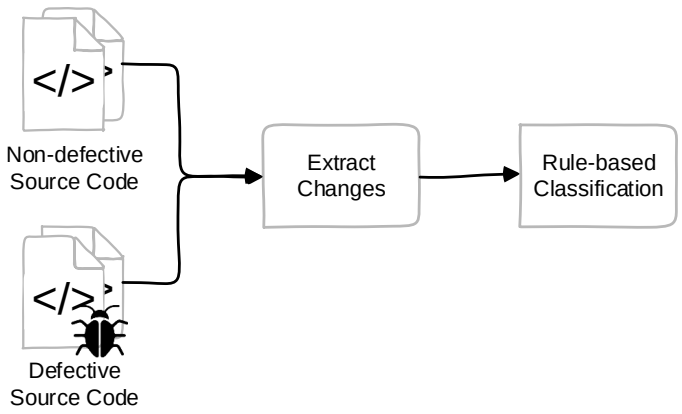


Figure 4.6.: Overview of our defect classification process.

data points for statistical analysis.

Figure 4.6 depicts our defect classification process. Overall the process has three steps. First, we need to gather the defective and non-defective version of the source code. In the second step, we extract the changes (i.e., the differences) between these two versions. Third, we apply our rule-based classification approach to assign a set of defect classes, which are presented in Figure 4.5.

As mentioned above, the input for our approach is the non-defective and the defective source code. Achieving the source code for the defects introduced by mutation analysis is complicated, as the chosen mutation testing framework (Section 4.2.6) does not offer the mutated source code, because the mutation is done on the byte-code of the original code [267]. Fortunately, some mutation operators that we have chosen (Section 4.2.6) can be directly mapped to a defect class, so we do not need to extract the changes beforehand. Table 4.6 shows all mutation operators together with their assigned defect class. The defect class is assigned in such a way, that they conform to our defect classification schema (Figure 4.5). For example, the "Null Return" operator replaces return values of function with `null`. It changes the return statements of methods, which is a sub category of the Logic/-Control class. Therefore, we can assign the Logic/Control class to all defects integrated via this mutation operator. Unfortunately, this does not work for all mutation operators (i.e., mutation operators marked with "-" in Table 4.6). For these mutation operators we applied the following approach. The mutation testing framework outputs the used mutation operator together with the line and file on which it was applied. We take the original source code of the affected file and integrate the defect, based on the rules of the mutation operator (Table 4.5), in the specified line. We do not know the *exact* change that was made by the mutation testing framework, as this information is not available. Nevertheless, the *exact* change is not needed to classify the integrated defect.

Mutation Operator	Defect Class
AP: *Argument Propagation*	Interface
BFR: *Boolean False Return*	Logic/Control
BTR: *Boolean True Return*	Logic/Control
CB: *Conditionals Boundary*	-
CC: *Constructor Calls*	Interface
EOR: *Empty Object Return*	Logic/Control
I: *Increments*	-
IC: *Inline Constant*	Data
IN: *Invert Negatives*	-
M: *Math*	-
MV: *Member Variable*	Computation
NR: *Naked Receiver*	Interface
NC: *Negate Conditionals*	-
NVMC: *Non Void Method Calls*	Interface
NR: *Null Return*	Logic/Control
PR: *Primitive Return*	Logic/Control
RC: *Remove Conditionals*	Logic/Control
RI: *Remove Increments*	-
RS: *Remove Switch*	Logic/Control
RV: *Return Values*	Logic/Control
S: *Switch*	Logic/Control
VMC: *Void Method Calls*	Interface

Table 4.6.: Mapping between the used mutation operators and the defect class.

```
1 public int g2(int i) {
2   if(i <= 0)
3     return 0;
4   else if(i==1)
5     return 1;
6   else
7     return g2(i-2)+g2(i-1);
8 }
```

Listing 4.1: Original source code (ex.).

```
1 public int g2(int i) {
2   if(i <= 0)
3     return 0;
4   else if(i==1)
5     return 1;
6   else
7     return g2(i-2)-g2(i-1);
8 }
```

Listing 4.2: Defective source code (ex.).

Consider the example given in listings 4.1 and 4.2. Listing 4.1 shows the original source code. Assume, that the math mutator (Table 4.5) was used in line 7 of the shown method. Using this information, we parse the original source code (shown in Listing 4.1) in line 7 to evaluate what kind of math operator is originally used in this line. In the given example

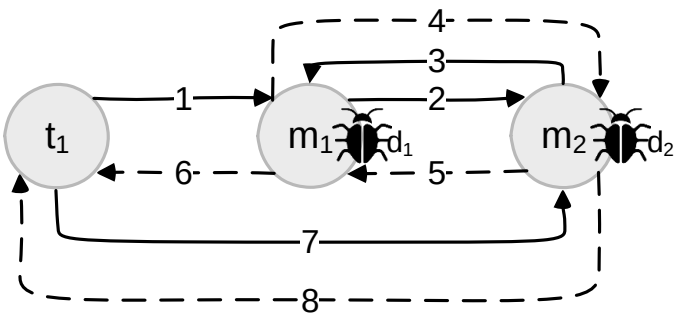


Figure 4.7.: Example call sequence graph of test t_1 together with two different methods (m_1, m_2) which contain defects (d_1, d_2). The number on the arrows indicate the order of the calls. The dashed arrows indicate return calls.

we determined, that the "+" operator is used. Afterwards, we change this operator via the rules of the math mutation operator (i.e., we change the "+" to a "-"). The resulting method is shown in Listing 4.2. Now, we have two variants of the source code that we can use for the next step of our process.

For the change extraction we adopted the tool CHANGEDISTILLER [268], which is able to extract fine-grained changes from Java source code to classify our changes. Basically, it extracts changes between two source code files by extracting and comparing the Abstract Syntax Trees (ASTs) of both files. For each change that is detected this way, CHANGEDISTILLER outputs the Change Type (CT) (as defined by Fluri and Gall [268]) together with the Changed Entity (CE) (e.g., an if-statement or a method), and the Parent Entity (PE) of the changed entity (e.g., the initialization part of a for loop).

The advantage that we gain is, that the CTs that can be extracted via CHANGEDISTILLER can be mapped to the defect classes presented in Figure 4.5. Table A.1 in Appendix A shows exactly this mapping. Unfortunately, not all CTs can be directly mapped onto a defect class. Hence, we need additional information (i.e., the CE and/or PE). Table A.2 in Appendix A highlights the conditions that need to be fulfilled for a change to be put into the corresponding defect class. These conditions were manually created by an incremental process. Every time a new change was found that could not be classified into a defect class, we manually classified the change and updated our rules accordingly.

4.2.8. Extracting Defect-Locality

Defect-locality is the metric that we use to evaluate, if the defect source detection is easier with unit than with integration tests. We define defect-locality as a function $dl : TxD \rightarrow \mathbb{N}$, where T is the set of all tests of a project, D is the set of all <, and $dl(t,d) =$ length of the **first** call sequence from t to a method m that contains the defect d, but only if the line in which d resides in m is covered by t.

Figure 4.7 illustrates an example call sequence graph of the test t_1 and two different methods (m_1, m_2). This figure depicts two defects in the program: d_1 in m_1 and d_2 in m_2. The arrows highlight the calls from the test case to a method or from a method to another method. Additionally, the numbers on the arrows depict the order of the calls. If we now assume that defect d_1 was not covered by the first call, but with the third (e.g., because only then the lines in which the defect resides in are executed) we get $dl(t_1, d_1) = 3$ as a result. If we assume that d_2 was covered by the seventh call and not by the second one, we get $dl(t_1, d_2) = 1$, as this is the **first** call sequence from the test case t_1 to the defect d_2 where d_2 was covered by t_1.

This approach simulates the call sequence that a developer would look through if he would want to debug a found defect. In general, the approach works by storing and updating the call sequence length during the execution of a test case. If the line in which a defect resides in is hit by the test case, we need to store the defect that was covered together with the current call sequence length at this point in time. The hard part is to find the line in which a defect resides in. Normally, we do not have this information available[8]. Therefore, we make use of the mutation detection results that we collect (Section 4.2.6). The mutation detection result of a test includes a list of mutants that were generated for the test, together with the line number and the file in which the mutant was introduced. It also includes if the mutant was covered and killed or survived the test, for each generated mutant. This data can then be used to insert the probes into the source code at the specific source code lines to check if (and with which call sequence length) a test covers a defect. Note, that we are only able to extract the defect-locality for Java projects, as we were not able to find a fitting mutation testing framework for Python (Section 4.2.6) and therefore, no defect information is available.

Other approaches like, e.g., calculating the shortest path from a test to the method that contains the defect in a call graph were not feasible. We want our data to be as precise as possible. If we would take the call graph as basis and calculate the shortest paths, we would ignore the fact that a method call only covers some lines in the code, but not necessarily the lines in which the defect resides in.

Our defect-locality metric has the limitation that it does not store the state of each unit that contains a method. Hence, it can happen that the first call to a method changed the state of the unit (e.g., setting a flag) and only because of this change the second call covered the defect (i.e., the defect gets executed). In this case a developer would not only debug the second call (like it is represented by the defect-locality metric), but also the first call to the unit. Nevertheless, the needed state information is not part of the execution trace itself. It would be possible to add more information to the trace (e.g., how the state of a unit was changed with a call), but this is out of scope for this thesis.

Another possibility to measure the defect-locality would be to count the overall number of calls till a defect is covered. For example, if the defect d_2 is covered with the seventh

[8] If this information would be available, the developers would have fixed the defect already.

CPU	Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz
RAM	16 GB
OS	Ubuntu 16.04
HDD/SSD	Samsung SSD 860 EVO mSATA 1TB

Table 4.7.: Specification of the laptop used to measure the execution time of tests.

call the value of this metric would be seven if we also count return calls, or four if not. The problem with this metric definition is that the number of calls is highly dependent on the project, which influences the comparability between projects. Furthermore, repeated method calls (e.g., if a method call is part of a loop) or multi-threading would skew this metric even more. Because of these reasons, we decided for the approach described above.

4.2.9. Execution Time Measurement

The extraction of the execution time of a test is straight forwards. We checkout the project release that we want to analyze. Afterwards, the tests are run using the test runner of the project. For Java projects, we just reuse the build management script that was available in the code of the project and execute the tests via this script. For Python projects, we had a look at the configuration file for the used continuous integration system to check how the tests should be executed. Afterwards, we replicated the steps to execute the tests. If the program runs on different Python versions (e.g., 2.7 and 3.5), we preferred the execution via Python 3.x. After the execution of the tests, the test runner, i.e JUnit [172] for Java projects and pytest [175] or nosetests [218] for Python projects, writes different result files containing the execution time of the executed tests. These result files are then parsed to extract the execution time of test. All execution times are then stored into the database. The specifications of the laptop on which the tests were executed is given in Table 4.7. We reduced the outside influences as much as possible by executing the tests in an isolated environment where the load of the laptop was as minimal as possible.

4.2.10. Implementation

In the following sections, we describe our implementations for our data collection approaches presented in Section 4.2. All of our implementations are developed as open-source software. The development of open-source software has several advantages, especially in a research context [269]. First, every researcher and practitioner has the possibility to reuse our code to advance the body of knowledge of software engineering. They do not need to re-implement approaches for their data collection. Second, projects that are open-source

are also transparent, i.e., people can assert all implementations. Proprietary software, on the other hand, often has the problem that there are, e.g., unforeseen limitations.

In Section 4.2.10.1 we present our SmartSHARK platform in detail, which implements the extraction of project meta-data (Section 4.2.3). The COllection of Metrics FOR Tests (COMFORT)-framework is presented in Section 4.2.10.1, which implements several of our data collection approaches (i.e., the extraction of the test level, TestLOC, pLOC, and the mutation detection capabilities). Finally, in Section 4.2.10.3 we present the design and implementation of a tool called DCD, which implements the collection of the defect-locality.

4.2.10.1. SmartSHARK

Empirical studies grow common in the field of software engineering. This trend is highlighted by hundreds of publications that were published in recent years. Nevertheless, we identified several problems within the state-of-the-art that threatens the replicability and validity of such empirical studies, i.e., the heavy re-use of data sets, the non-availability of data sets, the non-availability of implementations, the usage of small data sets, and the diverse tooling [270]. The SmartSHARK platform is an ecosystem that was build with the goal to counter those threats.

One part of the concept of SmartSHARK is the unified data collection process, which is shown in Figure 4.8. Basically, the researcher is able to provide data collection plugins, access results, and execute the data collection via the webserver. SmartSHARK is implemented as a plugin-architecture. All data collection programs are provided as plugin, which conform to certain interfaces. We designed SmartSHARK and its plugin interface in a way, that it has minimum restrictions for the plugins, as we want to support a broad range of different plugins. The execution of the data collection works by accessing the webserver, choosing the plugin and the project on which the plugin should run, and confirm the selection. Afterwards, the data collection plugin is triggered on a batch system, which can be, e.g., a High Performance Computing (HPC) system like in our current deployment[9]. The usage of a batch system has several advantages: first, it allows flexible data collection plugins, as the batch system is not fixed to one programming language. Second, it provides us with good scalability, as it enables us to collect data from more than one project at once (depending on the capability of the batch system). The batch system then stores the results in a MongoDB, which can then be accessed by the webserver or by any other client that has access rights. There are several reasons for using a MongoDB instead of a relational database like MySQL:

1. **flexibility:** the usage of a NoSQL database gives us more flexibility in storing and combining the data, as we do not have a fixed schema and can dynamically add or delete attributes [271].

[9] In our current deployment, we use the HPC system of the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG) (`https://www.gwdg.de`).

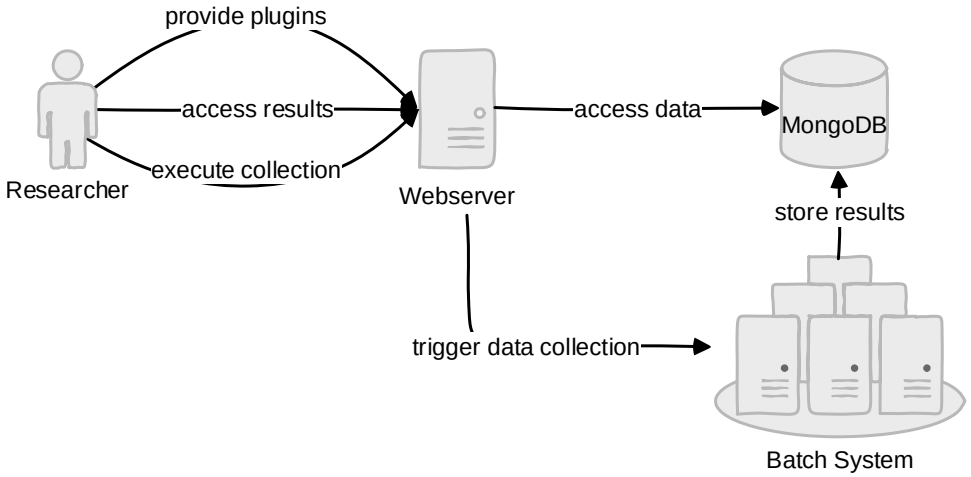


Figure 4.8.: Overview of the data collection part of SmartSHARK.

2. **established:** MongoDB itself is used and tested in big data environments, as we have in our SmartSHARK ecosystem [272].

3. **scalability:** MongoDB is highly scalable due to sharding, which is important for our use case where we potentially acquire TB of data [273].

4. **support:** MongoDB offers a good library support and documentation for different programming languages [274].

5. **redundancy and data availability:** MongoDB offers a replication functionality, which provides us with a redundant environment (i.e., replica sets [275]) that ensures a high data availability.

While we designed and developed several plugins for the SmartSHARK environment that are able to collect diverse data from software projects, we only explain one of them in more detail. This plugin, called vcsSHARK, is used in our study to collect the project meta-data (Section 4.2.3), which is then used to connect collected metrics to this project data to enable our analysis. Nevertheless, a full list of all developed plugins for SmartSHARK can be found in the Appendix B.1.

The vcsSHARK is a plugin that collects VCS data from git projects. This includes commits (with commit dates, messages, etc.), tags, and action on files together with the concrete textual change. The plugin is written in Python and uses the official libgit2 [276] library for collecting the data. Furthermore, to speed up the whole storage and parsing process it uses the multiprocessing library of Python [277], which starts several processes for parsing and storing the data.

Figure 4.9.: Phases of our COMFORT-Framework.

4.2.10.2. COMFORT-Framework

The COMFORT-framework [278] implements the approaches for collecting the test level (Section 4.2.4), collecting the TestLOC and pLOC (Section 4.2.5), and the approach for collecting the mutation detection capabilities presented in Section 4.2.6. The framework was build with the intention to provide researchers and practitioners with a framework that is able to collect different test-specific metrics. As we were not able to determine an existing framework that provides the capability to collect test-specific metrics with an extendable data collection and analysis process, we decided to build COMFORT.

Basically, COMFORT has four different phases, which are presented in Figure 4.9. Which parts of COMFORT are executed in which phase can be configured via a configuration file. In the following, we only go into detail of the parts of COMFORT that are relevant for the study described in this thesis. While we designed and implemented more than the parts of the framework we describe here, it would go beyond the scope of this section. Nevertheless, we list all different parts of COMFORT, separated by its phase, in Appendix B.2.

Loading the Data: The first phase of COMFORT is the data loading phase. In this phase the data structure, that is used in the following phases, is created. This data structure can be based on, e.g., a static call graph or a dependency graph. For our study presented in this thesis, we used the per-test coverage data as basis for the data structure. This data must be collected beforehand and stored in a file so that COMFORT can read out this file to create a data structure based on its contents.

Figure 4.10 gives an overview of our per-test coverage collection process. The basis of our coverage collection is Jacoco [279]. Jacoco is a coverage collection framework for Java programs. It supports the collection of coverage data on different levels of abstraction (e.g., class-level or method-level). We integrate Jacoco into the projects build process, i.e. by adding it to its maven build file, so that it produces the coverage data. We decided to use Jacoco as basis, as the development of an own coverage collection framework would be time-intensive and complex. Other advantages of Jaccoco are that it supports all major Java versions [280], that it can be integrated into the build process of Java projects [281], provides an API [282], and has a good documentation and support [283].

Nevertheless, Jacoco does not collect per-test coverage, but test suite coverage. Hence, we need to intercept the coverage collection process, so that a new coverage collection session is started before each test run. This is the idea of the comfort-listener [284]. The

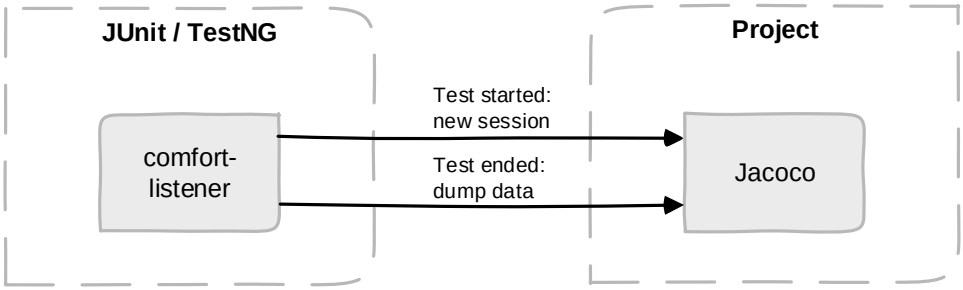


Figure 4.10.: Overview of the per-test coverage collection.

comfort-listener makes use of the JUnit [172]/TestNG [173] API and can be integrated by adding a custom test listener via the Maven SureFire plugin [285]. Before a test is started, a new coverage collection session is initiated by using the Jacoco API. After a test finished, the session is dumped into a file. This way, the coverage that is stored in the file and later on read by the COMFORT framework is a per-test coverage. Note, that we do not collect the coverage that might be generated by calling potential setup or break-down methods [286], as we focus on the coverage of the test itself.

While Figure 4.10 shows the per-test coverage collection for Java projects, the collection process looks the same for Python projects. Instead of Jacoco, we reuse coverage.py [287] as coverage collection framework and instead of JUnit/TestNG as test runner we use unittest [174]/pytest [175]. The comfort-smother [288] replaces the comfort-listener for Python projects.

Filtering the Data: After the data is loaded and the data structure is created, we can apply different filters on the data structure, e.g. every covered unit that is not part of the project should be excluded. As we do not apply any filter in the study presented in this thesis, we do not describe them in detail.

Collecting Metrics: In the third phase, the data structure is given to the metric collectors that were configured via the configuration file. Overall, COMFORT provides 13 different metric collectors that work with Java/Python projects. Nevertheless, we only describe the ones that are used in our study.

- **IEEETestTypeCollector:** Detects the test type (i.e., unit or integration test) using the IEEE rule set presented in Table 4.3. Hence, it implements the test level classification approach described in Section 4.2.4 for the IEEE definition.
- **ISTQBTestTypeCollector:** Detects the test type (i.e., unit or integration test) using the ISTQB rule set presented in Table 4.3. Hence, it implements the test level classification approach described in Section 4.2.4 for the ISTQB definition.

- **NamingConventionTestTypeCollector:** Detects the test type (i.e., unit or integration test) using the DEV rule set presented in Table 4.3. Hence, it implements the test level classification approach described in Section 4.2.4 for the developer classification.
- **CoveredLinesCollector:** Collects the TestLOC and pLOC for each test that is part of the loaded data. It implements the approach described in Section 4.2.5.
- **MutationDataCollector:** Collects the mutation detection capabilities of each test that is part of the loaded data. The MutationDataCollector implements the approach described in Section 4.2.6. Hence, it uses PIT [267] to generate mutants and collect the mutation results. Afterwards, it parses the results generated by PIT and classifies all generated mutants into its corresponding defect class according to the approach presented in Section 4.2.7.

Storing the Data: In the last phase, the collection results are stored using a filer. As we want to interconnect the results from COMFORT with the project meta-data collected by SmartSHARK, we created a SmartSHARK filer. This filer is able to store and interconnect the collected data with data collected by SmartSHARK plugins.

4.2.10.3. Defect Call Depth (DCD)

The approach to collect the defect-locality of tests (Section 4.2.8) is implemented in a tool called DCD. Figure 4.11 gives an overview of how DCD works. We explain each step that is marked in the figure in the following.

1. As a first step, we integrate the dcd-agent into the project. The dcd-agent is an implementation of a Java agent [34] that uses the instrumentation API of Java. It is responsible for integrating method calls into the project classes (Step 4). The integration is done by using the "-javaagent" parameter of the Java Virtual Machine (JVM). This parameter is added to the Maven SureFire plugin execution [119] so that our agent gets executed with the tests of the project.
2. As a second step, we integrate the dcd-listener. The dcd-listener is similar to the comfort-listener explained in Section 4.2.10.2: it also makes use of the JUnit [172] API and is integrated by adding it as a custom test listener to the Maven SureFire plugin [285].
3. After everything is integrated, and the test process via the build management system is started by hand, the dcd-agent queries the mutation data from the MongoDB of the SmartSHARK environment. The mutation data includes information about which test covered which mutant together with the file name and the line in which the mutant was integrated.

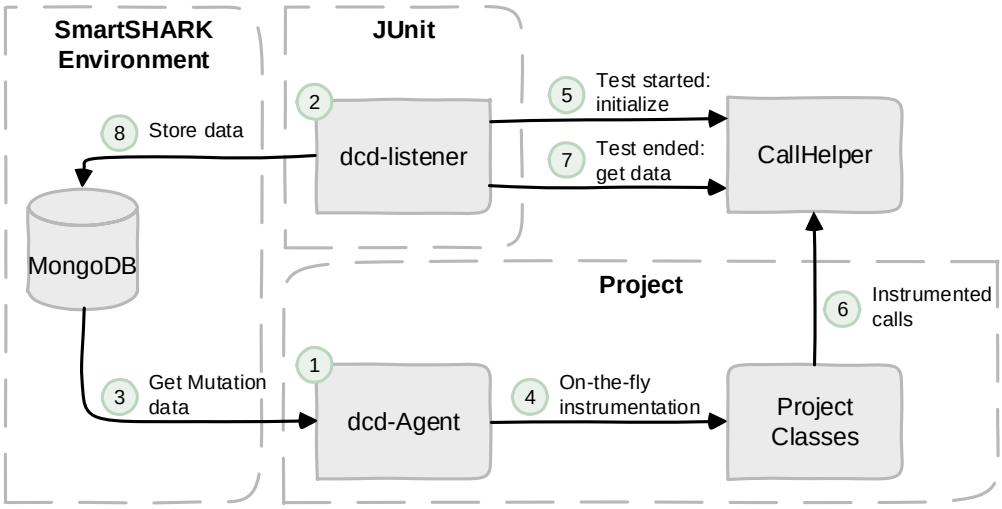


Figure 4.11.: Overview of DCD.

4. Afterwards, the dcd-agent does an on-the-fly instrumentation of the project classes. The on-the-fly instrumentation "allows in-memory pre-processing of all class files during class loading independent of the application framework." [289]. While there are other instrumentation techniques, we decided for on-the-fly instrumentation due to the following reasons. First, it is a byte code based instrumentation technique. Hence, it "is very fast, it can be implemented in pure Java and works with every Java VM" [290]. Second, using on-the-fly instrumentation enables us to do modifications to the loaded classes without modifying the target application itself.

Using on-the-fly instrumentation the dcd-agent integrates several method calls to the CallHelper class into the project classes that are loaded. The CallHelper class is a static class that holds information (e.g., the current call depth, which represents the defect-locality) during the whole testing process. Before each method call in any project class, we integrate a call to the CallHelper that raises the call depth. After each method call, the call depth is lowered. In addition, we integrate a call at each position were a mutant was integrated before during the collection of the mutation detection capabilities (Section 4.2.6). However, within some tests exceptions are thrown and caught via the try-catch construct in Java [122]. If this is the case, the injected method call to lower the depth after a method is never executed. To mitigate this problem, we add a call to lower the depth within each catch block.

5. If the test is started, the dcd-listener initializes the CallHelper, i.e., it sets the current call depth to zero.

6. Now the test gets executed including its setup and break-down methods [286], as

they are also executed during the collection of the mutation detection capabilities. During its run it executes the instrumented calls to the CallHelper, i.e., the current call depth is raised and lowered. If the test hits a call at a position where a mutant was integrated before, we store the file name together with the line number in which the call was integrated and the current call depth in a map in the CallHelper. This map then holds information about the covered files and lines together with the call depth that was present when the test hit the call.

7. After the test finished, the dcd-listener queries the data from the CallHelper, i.e., the map that includes the call depth for each file and line that was hit by the test.

8. Finally, the map is parsed and stored into the MongoDB and interconnected with the currently available results.

4.2.11. Qualitative Data Collection

Within this thesis we want to gain qualitative evidence to answer our RQs 2.4 - 2.6. We could not gain quantitative evidence for these RQs out of different reasons.

RQ 2.4 is concerned with the test execution automation of unit and integration tests. The standard literature states that unit tests are easily automatable. Therefore, we want to evaluate, how unit and integration tests are executed nowadays. While our quantitative analysis can help in answering this RQ, e.g., by providing evidence what kind of tests are automatically executed within our case study subjects, it can not be completely answered by it. We can not define a fitting metric that would cover the aspects of this RQ. Hence, we analyze related literature that has contributed to the topic of test automation to assess, which tests (i.e., unit or integration tests) can be automated. We want to gain an understanding of the actual usage of test automation tools and for which tests they are practically used. Therefore, we do not only evaluate the research view, but also the practical view by including, e.g., developer comments, developed frameworks, or the industrial landscape in our qualitative analysis of this RQ. The qualitative evidence that we gain throughout this analysis is then used to answer RQ 2.4.

RQ 2.5 is concerned with the different test objectives of unit and integration tests. Spillner et al. [9] state that unit tests should be used to test the efficiency, maintainability, and robustness of a software. We acquire qualitative evidence on the usage of unit and integration tests for those testing types. The procedure that we use to gain this evidence is similar to the procedure explained above. We assess this question from the research and practical view by evaluating related research and including developer comments and frameworks that were developed for those specific tests. In the end, we gain a holistic view on RQ 2.5 and provide qualitative evidence to answer our RQ regarding the differences in the test objective.

RQ 2.6 is concerned with the difference in costs of unit and integration tests. We found a statement in the standard literature that unit tests cost less than integration tests. While this could be measured by a metric (e.g., creation costs of tests or maintenance costs) we do not

have this data available, as we use open-source projects for our case study. We decided to collect qualitative evidence on the costs of tests on different levels, by evaluating research literature that is related to test costs. Furthermore, we want to include evidence provided by the industry within this topic. We include, e.g., comments or blog posts by developers and companies that discuss the costs of unit and/or integration tests. By evaluating the collected evidence we can answer the question, if unit or integration tests are more costly.

4.3. Data Analysis

Figure 4.1 shows the steps taken in our study. Our study consisted of the following steps:

- An analysis of the distribution of unit and integration tests in open-source projects (Section 5)
- A quantitative evaluation of the differences between unit and integration tests (Section 6)
- A qualitative evaluation of the differences between unit and integration tests (Section 7)

Due to the diversity of our approaches to analyze the data (e.g., qualitative and quantitative analysis) and the different foci of our RQs, we refrain from describing the analysis in detail in this section. The data analysis and the results of the analysis are described in the different sections referenced above. During our analysis we need to perform several statistical tests (Section 2.3). Following the process defined in Section 2.3.2, we need to state our significance level for the statistical tests that we perform. During this thesis, we follow the advise by Benjamin et al. [98] for all of our statistical tests and set our significance level to .005, i.e. $\alpha = .005$.

5. Distribution of Unit and Integration Tests in Open-Source Projects

Within this section, we present the data set that we used within the analysis of our first RQ (Section 5.1) together with the descriptions of the analysis of each sub question (sections 5.2- 5.4). The analysis sections include the analysis procedure, as well as the results of the analysis, together with a concrete answer to the RQ at hand.

5.1. Data Set Description

The data set used to answer the first RQ, together with its sub questions, includes all tests of all selected projects (Section 4.2.2). The data set was generated in the following way. For each subject project, we executed the tests after we implemented our listener(s) to record the per-test coverage.

The test execution for all Java projects was done by using the *test* goal of the Maven build management system and using Java 8. Hence, we reuse the build file of the developers of the projects and therefore tests that are excluded by the developers (e.g., because they are currently not working) are not executed. Moreover, for projects that have several submodules (i.e., *guice*, *google-gson*, and *zxing*), we only analyzed the core of the project, as generating coverage for multi-module projects is out of scope for this thesis.

For Python projects we do not have such a build management system that we can reuse. Hence, we reproduced the steps that are taken by the CI system of the projects to test the project. Therefore, if tests are excluded for Python projects by the developers only, they are not executed. If the Python project works with Python 2.7 and Python 3.x, we always used the Python 3.x version.

For both the Java and Python projects, we excluded all tests that threw an error during the execution, as they would produce incorrect coverage results. This is the case, because only the coverage until the error occurrence can be recorded. We would have done the same with failed tests due to the same reason, but during our data set creation no tests failed.

After we recorded the per-test coverage data we executed the metric collectors of our COMFORT framework (Section 4.2.10.2) to read in the data and execute the different metric collectors, including the one that assigns a test level to each test. Parameterized tests are stored in a special way: as they might cover different parts of the program, we store one test entity for each parameter. For example, if the test *MisfitsTest* is called with three parameters (*a*, *b*, *c*), we store three different test entities in the database: *MistfitsTest[a]*,

Project	#All Tests	#Analyzed Tests
commons-beanutils	1197	1175
commons-codec	874	853
commons-collections	6637	5930
commons-io	1150	1138
commons-lang	4064	3978
commons-math	6488	6484
druid	4132	4131
fastjson	4176	4157
google-gson	1014	1012
guice	701	701
HikariCP	120	118
jackson-core	774	774
jfreechart	2175	2174
joda-time	4176	4154
jsoup	593	589
mybatis-3	1053	1048
zxing	401	401
ChatterBot	299	283
csvkit	177	177
dpark	58	58
mrjob	1862	1836
networkx	2723	2672
pyramid	2626	2536
python-telegram-bot	638	627
rq	207	203
schematics	348	343
scrapy	1656	1591
Mean	1863.67	1820.11
StDev	1901.95	1831.26

Table 5.1.: Projects together with the number of all and analyzed tests.

MisfitstsTest[b], and *MisfitsTest[c]*. The coverage framework already separates the test, so that we do not need to manually interfere.

Table 5.1 summarizes the number of all the tests and analyzed tests for each subject project. Tests are filtered out, e.g., when they are empty. Our filter criteria are mentioned in Section 4.2.4.

For the analysis of our RQs, we need the test level of each test for each subject project according to the ISTQB, IEEE (RQ 1.1, RQ 1.3), and the DEV (RQ 1.2) rule sets. We define the set of unit tests as U and the set of integration tests as I. A set of unit or integration tests for a specific rule set is defined as U_x and I_x where $x \in \{DEV, IEEE, ISTQB\}$. A project specific set of unit or integration tests for a specific rule set is defined as $U_x(p)$ and $I_x(p)$ where $x \in \{DEV, IEEE, ISTQB\}$ and $p \in Projects$, where projects is the set of all subject projects. In addition to the test level, we need the TestLOC of each test for the subsequent

analysis. Hence, we define a function $tl(X) := \sum_{x \in X} TestLOC(x)$, where X is a set of tests and $TestLOC(x)$ is the number of TestLOC for the specific test x. Additionally, we acquired the KLOC of all projects using sloccount [291], a command line tool that counts the LOC of different languages. We applied it to the folder of the projects that contain the source code (production and test code) and used the reported number later for normalization.

Table 5.2 presents the subject projects, together with their KLOC and the number of tests within the different test sets. This table also highlights the percentages of tests that are contained in a test set for a specific definition. Furthermore, Table 5.2 includes the mean and the standard deviation for the different test sets across all subject projects.

Table 5.3 presents the subject projects, together with the tl metric within the analyzed test sets. Furthermore, the percentages for the tl metric for each test set are highlighted. Table 5.3 includes the mean and the standard deviation for the tl metric.

5.2. Evaluation of RQ 1.1: Test Distribution Trends

This RQ is concerned with the trend mentioned in the introduction, i.e. that more integration than unit tests are developed, and whether it is visible in actual versions of open-source projects. This section describes the analysis procedure for RQ 1.1 (Section 5.2.1), as well as the results of the analysis (Section 5.2.2).

5.2.1. Analysis Procedure

For the assessment of this RQ, we use the test level classification for the IEEE and ISTQB rule sets. After gathering the test levels of all tests, we check whether there are differences between the amount of tests at both test levels. We decided to use two different metrics to assess this RQ: the number of tests and the tl. While the first metric is rather coarse, the second metric presents a more detailed view on the tests and considers the test sizes. As we have chosen the projects randomly from a list of fitting projects (Section 4.2.2) we have, by the nature of this procedure, a large variance between the projects in terms of the selected metrics. According to tables 5.2 and 5.3, larger projects (in terms of KLOC) tend to have a larger number of tests and correspondingly larger tl. Therefore, to enable a comparison between the projects we need to account for the aforementioned problem by normalizing the chosen metrics. To assess and measure the differences after these preparations, we use statistical tests.

Our analysis process is described in detail in the following:

1. **Gather the Data:** We gather the tests with their classification results that were created via the IEEE and ISTQB rule sets, resulting in the $U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}$ test sets for all subject projects.

2. **Normalization:** Afterwards, we perform two different normalizations, once for the number of tests and once for the tl metric. Hence, we define two different metrics:

Project	KLOC	$\|U_{IEEE}\|$	(%)	$\|I_{IEEE}\|$	(%)	$\|U_{ISTQB}\|$	(%)	$\|I_{ISTQB}\|$	(%)	$\|U_{DEV}\|$	(%)	$\|I_{DEV}\|$	(%)
commons-beanutils	33.421	285	(24.26)	890	(75.74)	44	(3.74)	1131	(96.26)	0	(0.00)	1175	(100.00)
commons-codec	19.883	707	(82.88)	146	(17.12)	343	(40.21)	510	(59.79)	715	(83.82)	138	(16.18)
commons-collections	60.733	1925	(32.46)	4005	(67.54)	707	(11.92)	5223	(88.08)	5021	(84.67)	909	(15.33)
commons-io	29.150	634	(55.71)	504	(44.29)	240	(21.09)	898	(78.91)	428	(37.61)	710	(62.39)
commons-lang	77.224	3507	(88.16)	471	(11.84)	1671	(42.01)	2307	(57.99)	2503	(62.92)	1475	(37.08)
commons-math	208.840	775	(11.95)	5709	(88.05)	376	(5.80)	6108	(94.20)	5429	(83.73)	1055	(16.27)
druid	236.249	94	(2.28)	4037	(97.72)	83	(2.01)	4048	(97.99)	298	(7.21)	3833	(92.79)
fastjson	146.439	315	(7.58)	3842	(92.42)	97	(2.33)	4060	(97.67)	291	(7.00)	3866	(93.00)
google-gson	21.547	215	(21.25)	797	(78.75)	131	(12.94)	881	(87.06)	291	(28.75)	721	(71.25)
guice	30.549	28	(3.99)	673	(96.01)	7	(1.00)	694	(99.00)	169	(24.11)	532	(75.89)
HikariCP	11.334	21	(17.80)	97	(82.20)	16	(13.56)	102	(86.44)	17	(14.41)	101	(85.59)
jackson-core	41.602	47	(6.07)	727	(93.93)	30	(3.88)	744	(96.12)	89	(11.50)	685	(88.50)
jfreechart	134.117	228	(10.49)	1946	(89.51)	169	(7.77)	2005	(92.23)	2105	(96.83)	69	(3.17)
joda-time	85.485	179	(4.31)	3975	(95.69)	55	(1.32)	4099	(98.68)	1378	(33.17)	2776	(66.83)
jsoup	18.782	64	(10.87)	525	(89.13)	46	(7.81)	543	(92.19)	401	(68.08)	188	(31.92)
mybatis-3	48.974	224	(21.37)	824	(78.63)	79	(7.54)	969	(92.46)	431	(41.13)	617	(58.87)
zxing	31.772	108	(26.93)	293	(73.07)	61	(15.21)	340	(84.79)	34	(8.48)	367	(91.52)
ChatterBot	7.532	59	(20.85)	224	(79.15)	43	(15.19)	240	(84.81)	91	(32.16)	192	(67.84)
csvkit	2.857	36	(20.34)	141	(79.66)	29	(16.38)	148	(83.62)	177	(100.00)	0	(0.00)
dpark	12.576	18	(31.03)	40	(68.97)	13	(22.41)	45	(77.59)	57	(98.28)	1	(1.72)
mrjob	36.900	912	(49.67)	924	(50.33)	470	(25.60)	1366	(74.40)	1677	(91.34)	159	(8.66)
networkx	59.890	779	(29.15)	1893	(70.85)	357	(13.36)	2315	(86.64)	1379	(51.61)	1293	(48.39)
pyramid	43.480	1207	(47.59)	1329	(52.41)	692	(27.29)	1844	(72.71)	2201	(86.79)	335	(13.21)
python-telegram-bot	18.954	238	(37.96)	389	(62.04)	180	(28.71)	447	(71.29)	508	(81.02)	119	(18.98)
rq	5.453	45	(22.17)	158	(77.83)	8	(3.94)	195	(96.06)	182	(89.66)	21	(10.34)
schematics	7.885	86	(25.07)	257	(74.93)	60	(17.49)	283	(82.51)	74	(23.20)	245	(76.80)
scrapy	25.560	418	(26.27)	1173	(73.73)	400	(25.14)	1191	(74.86)	434	(27.28)	1157	(72.72)
Mean	53.97	487.19	27.35	1332.93	72.65	237.30	14.65	1582.81	85.35	977.04	50.92	842.19	49.08
StDev	60.36	750.10	21.74	1558.56	21.74	350.54	11.37	1686.13	11.37	1426.78	34.21	1061.74	34.21

Table 5.2.: KLOC, number and percentage of tests in the different test sets for the selected projects.

Project	$tl(U_{IEEE})$	(%)	$tl(I_{IEEE})$	(%)	$tl(U_{ISTQB})$	(%)	$tl(I_{ISTQB})$	(%)	$tl(U_{DEV})$	(%)	$tl(I_{DEV})$	(%)
commons-beanutils	10966	(19.77)	44510	(80.23)	938	(1.69)	54538	(98.31)	0	(0.00)	55476	(100.00)
commons-codec	8336	(82.52)	1766	(17.48)	3804	(37.66)	6298	(62.34)	8196	(81.13)	1906	(18.87)
commons-collections	63048	(19.46)	260958	(80.54)	15767	(4.87)	308239	(95.13)	242934	(74.98)	81072	(25.02)
commons-io	17345	(54.67)	14381	(45.33)	3324	(10.48)	28402	(89.52)	8196	(25.83)	23530	(74.17)
commons-lang	48268	(81.66)	10841	(18.34)	19312	(32.67)	39797	(67.33)	38817	(65.67)	20292	(34.33)
commons-math	12009	(6.98)	160103	(93.02)	4791	(2.78)	167321	(97.22)	143453	(83.35)	28659	(16.65)
druid	585	(0.81)	71597	(99.19)	455	(0.63)	71727	(99.37)	6266	(8.68)	65916	(91.32)
fastjson	1973	(4.55)	41386	(95.45)	621	(1.43)	42738	(98.57)	2816	(6.49)	40543	(93.51)
google-gson	2185	(16.43)	11115	(83.57)	1298	(9.76)	12002	(90.24)	2775	(20.86)	10525	(79.14)
guice	265	(1.90)	13714	(98.10)	57	(0.41)	13922	(99.59)	3483	(24.92)	10496	(75.08)
HikariCP	318	(5.16)	5848	(94.84)	147	(2.38)	6019	(97.62)	444	(7.20)	5722	(92.80)
jackson-core	490	(1.99)	24086	(98.01)	328	(1.33)	24248	(98.67)	1861	(7.57)	22715	(92.43)
jfreechart	2642	(7.80)	31240	(92.20)	1974	(5.83)	31908	(94.17)	32719	(96.57)	1163	(3.43)
joda-time	2810	(3.23)	84122	(96.77)	479	(0.55)	86453	(99.45)	26747	(30.77)	60185	(69.23)
jsoup	705	(5.64)	11795	(94.36)	485	(3.88)	12015	(96.12)	8156	(65.25)	4344	(34.75)
mybatis-3	1449	(4.81)	28663	(95.19)	518	(1.72)	29594	(98.28)	9956	(33.06)	20156	(66.94)
zxing	1564	(10.17)	13809	(89.83)	713	(4.64)	14660	(95.36)	796	(5.18)	14577	(94.82)
ChatterBot	349	(9.95)	3158	(90.05)	273	(7.78)	3234	(92.22)	597	(17.02)	2910	(82.98)
csvkit	345	(16.76)	1714	(83.24)	285	(13.84)	1774	(86.16)	2059	(100.00)	0	(0.00)
dpark	221	(14.64)	1289	(85.36)	146	(9.67)	1364	(90.33)	1498	(99.21)	12	(0.79)
mrjob	15452	(5.57)	262098	(94.43)	4101	(1.48)	273449	(98.52)	177815	(64.07)	99735	(35.93)
networkx	17084	(26.60)	47148	(73.40)	6734	(10.48)	57498	(89.52)	31888	(49.65)	32344	(50.35)
pyramid	22686	(28.45)	57057	(71.55)	9251	(11.60)	70492	(88.40)	72208	(90.55)	7535	(9.45)
python-telegram-bot	2012	(28.42)	5067	(71.58)	1225	(17.30)	5854	(82.70)	6237	(88.11)	842	(11.89)
rq	373	(14.79)	2149	(85.21)	46	(1.82)	2476	(98.18)	2207	(87.51)	315	(12.49)
schematics	508	(8.04)	5814	(91.96)	393	(6.22)	5929	(93.78)	1010	(15.98)	5312	(84.02)
scrapy	3282	(12.87)	22211	(87.13)	3092	(12.13)	22401	(87.87)	4433	(17.39)	21060	(82.61)
Mean	8787.78	(18.28)	45838.48	(81.72)	2983.59	(7.96)	51642.67	(92.04)	31021.00	(46.93)	23605.26	(53.07)
StDev	15120.93	(21.67)	70988.53	(21.67)	4792.80	(9.15)	77882.33	(9.15)	60509.96	(35.35)	26994.69	(35.35)

Table 5.3.: Number and percentage of *tl* in the different test sets for the selected projects.

$nm_C(X,p) := \frac{|X(p)|}{KLOC(p)}$ to normalize the number of tests and $nm_{TL}(X,p) := \frac{tl(X(p))}{KLOC(p)}$ to normalize the TestLOC sums, where test set $X \in \{U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}\}$ and p is a project. Furthermore, we define the set of all nm_C metric values of all projects as $NM_C(X) := \{nm_C(X,p) | p \in Projects\}$ and the set of all nm_{TL} metric values of all projects as $NM_{TL}(X) := \{nm_{TL}(X,p) | p \in Projects\}$, where test set $X \in \{U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}\}$ and p is a project. As a result, we get values for both normalized metrics across the four test sets.

3. **Check Preconditions for Statistical Testing:** We separately check for each of the $NM_C(X)$ and $NM_{TL}(X)$ sets if their values follow a normal distribution using the Shapiro-Wilk test (Section 2.3.5.1). Moreover, we separately test for equal variances using the Brown-Forsythe test (Section 2.3.5.2) between $NM_C(U_{IEEE})$ and $NM_C(I_{IEEE})$, as well as between $NM_C(U_{ISTQB})$ and $NM_C(I_{ISTQB})$. Furthermore we perform the same statistical test for the $NM_{TL}(X)$ sets analogously. These tests are necessary in order to choose the correct statistical significance test in the next step.

4. **Statistical Testing:** Based on the results of the previous step, we chose a fitting one-sided significance test (Section 2.3.1) to test for differences between $NM_C(U_{IEEE})$ and $NM_C(I_{IEEE})$, as well as between $NM_C(U_{ISTQB})$ and $NM_C(I_{ISTQB})$. Furthermore, we test for significant differences using the $NM_{TL}(X)$ sets analogously. We have chosen a one-sided test, because we want to assess if the normalized metrics of the unit tests sets (i.e., the normalized number of unit tests or their TestLOC) are significantly smaller than the normalized metrics of the other sets. A two-sided test would only tell us that there are differences, but not in which direction.

5. **Effect Size:** Finally, if we found a statistically significant difference between the normalized metrics of the sets of unit and integration tests, we assess the effect size by calculating Cohen's d (Section 2.3.6) between them to measure the practical relevance of the observed results.

5.2.2. Results

Table 5.4 shows the nm_C and nm_{TL} metrics for each project and for both rule sets. Overall, there are 24 (NM_C and NM_{TL}, IEEE) and 27 (NM_C and NM_{TL}, ISTQB) of overall 27 projects, which have less unit than integration tests. Even the projects, which have more unit than integration tests (i.e., *commons-codec*, *commons-io*, *commons-lang*) are the same. This is also highlighted by the mean that was calculated for both metrics. Table 5.4 shows that the mean for both metrics and definitions is lower for the unit test sets than the mean for the integration test sets, while the standard deviation is high for the nm_{TL} metric for both definitions. This gives a first hint that there are indeed less unit than integration tests developed (measured in numbers and TestLOC).

Project	$nm_C(U_{IEEE})$	$nm_C(I_{IEEE})$	$nm_C(U_{ISTQB})$	$nm_C(I_{ISTQB})$	$nm_{TL}(U_{IEEE})$	$nm_{TL}(I_{IEEE})$	$nm_{TL}(U_{ISTQB})$	$nm_{TL}(I_{ISTQB})$
commons-beanutils	8.53	26.63	1.32	33.84	328.12	1331.80	28.07	1631.85
commons-codec	35.56	7.34	17.25	25.65	419.25	88.82	191.32	316.75
commons-collections	31.70	65.94	11.64	86.00	1038.12	4296.81	259.61	5075.31
commons-io	21.75	17.29	8.23	30.81	595.03	493.34	114.03	974.34
commons-lang	45.41	6.10	21.64	29.87	625.04	140.38	250.08	515.34
commons-math	3.71	27.34	1.80	29.25	57.50	766.63	22.94	801.19
druid	0.40	17.09	0.35	17.13	2.48	303.06	1.93	303.61
fastjson	2.15	26.24	0.66	27.72	13.47	282.62	4.24	291.85
google-gson	9.98	36.99	6.08	40.89	101.41	515.85	60.24	557.01
guice	0.92	22.03	0.23	22.72	8.67	448.92	1.87	455.73
HikariCP	1.85	8.56	1.41	9.00	28.06	515.97	12.97	531.06
jackson-core	1.13	17.48	0.72	17.88	11.78	578.96	7.88	582.86
jfreechart	1.70	14.51	1.26	14.95	19.70	232.93	14.72	237.91
joda-time	2.09	46.50	0.64	47.95	32.87	984.06	5.60	1011.32
jsoup	3.41	27.95	2.45	28.91	37.54	627.99	25.82	639.71
mybatis-3	4.57	16.83	1.61	19.79	29.59	585.27	10.58	604.28
zxing	3.40	9.22	1.92	10.70	49.23	434.63	22.44	461.41
ChatterBot	7.83	29.74	5.71	31.86	46.34	419.28	36.25	429.37
csvkit	12.60	49.35	10.15	51.80	120.76	599.93	99.75	620.93
dpark	1.43	3.18	1.03	3.58	17.57	102.50	11.61	108.46
mrjob	24.72	25.04	12.74	37.02	418.75	7102.93	111.14	7410.54
networkx	13.01	31.61	5.96	38.65	285.26	787.24	112.44	960.06
pyramid	27.76	30.57	15.92	42.41	521.76	1312.26	212.76	1621.25
python-telegram-bot	12.56	20.52	9.50	23.58	106.15	267.33	64.63	308.85
rq	8.25	28.97	1.47	35.76	68.40	394.09	8.44	454.06
schematics	10.91	32.59	7.61	35.89	64.43	737.35	49.84	751.93
scrapy	16.35	45.89	15.65	46.60	128.40	868.97	120.97	876.41
Mean	11.62	25.61	6.11	31.12	191.69	934.07	68.97	1056.79
StDev	12.08	14.60	6.21	16.33	257.00	1462.24	78.66	1572.50

Table 5.4.: Normalized test count values (nm_C) and normalized tl values (nm_{TL}) for each project.

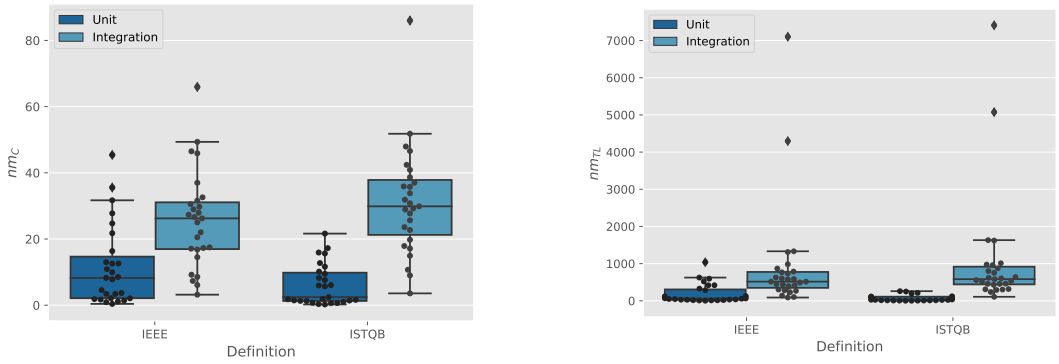


Figure 5.1.: Box-plot of the nm_C metric (left) and nm_{TL} metric (right) for unit and integration tests and the IEEE and ISTQB definition. The points in the plot represent the concrete values for each project.

Figure 5.1 shows box-plots of the nm_C (left) and the nm_{TL} (right) metrics. If we compare the box plots visually, we can determine a difference in them for each metric and for each definition. The box-plots highlight that the median is always lower for unit tests than integration tests, regardless of the used definition or metric. Furthermore, the 25%-quantile of the integration test box-plot is always higher than the 75%-quantile of the unit test box-plot. This indicates that the trend of developing less unit than integration tests is visible in our data.

While we see a difference in the data (Table 5.4) and visually (Figure 5.1) we do not know if the differences are significant. Hence, we performed several statistical tests. The concrete p-values and test statistics for each statistical test are reported in Appendix C. The Shapiro-Wilk tests showed that the NM_C set for the integration tests of both definitions follow a normal distribution, while all other tested sets do not. The Brown-Forsythe test showed, that all tested sets are homoscedastic for the chosen alpha level (i.e., $\alpha = .005$). The one-sided Mann-Whitney-U tests were performed to determine whether there is a significant difference between the metrics based on the unit test sets and the ones which are based on the integration test sets. The U-statistic was significant at the .005 critical alpha level, for all tested sets. Therefore, we reject H_0 for all of them and conclude that the difference was significant and that the projects have a larger amount of integration tests, both in terms of number of tests and tl for both definitions.

To assess the effect size we applied Cohen's d to all sets, which showed statistically significant differences. We measured a large effect ($d = 1.04$) between $NM_C(U_{IEEE})$ and $NM_C(I_{IEEE})$, a huge effect ($d = 2.02$) between $NM_C(U_{ISTQB})$ and $NM_C(I_{ISTQB})$, a medium effect ($d = 0.71$) between $NM_{TL}(U_{IEEE})$ and $NM_{TL}(I_{IEEE})$, and a large effect ($d = 0.89$) between $NM_{TL}(U_{ISTQB})$ and $NM_{TL}(I_{ISTQB})$.

Answer to RQ 1.1: Table 5.4 and Figure 5.1 highlight that there are differences in the median and mean between unit and integration tests (measured in nm_C and nm_{TL}). A high standard deviation for the nm_{TL} metric indicates a high variance in this metric among the (selected) projects. To prove the differences that are visible through the figures and the table we performed several statistical tests, which all showed that there is a larger amount of integration tests (measured in nm_C and nm_{TL}) for both definitions. Furthermore, the effect is not negligible and practically relevant, as our effect size measurements show between medium and huge effect sizes.

Overall, based on the data, visualizations, and tests we can conclude that the trend of developing less unit than integration tests is visible in open-source software projects.

5.3. Evaluation of RQ 1.2: Test Distribution according to Developer Classification

This RQ is similar to RQ 1.1, but tries to evaluate the trend that there are more integration than unit tests developed, from a different perspective. Within this RQ, we want to assess if the test level classification done by the developers are showing the trend explained in the introduction. This section summarizes our analysis procedure (Section 5.3.1), as well as the results of the analysis (Section 5.3.2).

5.3.1. Analysis Procedure

For the assessment of this RQ, we acquire the test level classification for the DEV rule set of each test that we analyze. Afterwards, we perform the same analysis as presented in Section 5.2.1, including the normalization due to the same reasons.

Our analysis process is described in detail in the following:

1. **Gather the Data:** We gather the tests with their classification results that were created via the DEV rule set, i.e., we query U_{DEV} and I_{DEV} from the database. Furthermore, we query the sum of TestLOC for these sets, i.e., $tl(U_{DEV})$ and $tl(I_{DEV})$.

2. **Normalization:** Afterwards we perform the same normalization and create the same metric sets as explained in Section 5.2.1. Hence, we create the $NM_C(U_{DEV})$, $NM_C(I_{DEV})$, $NM_{TL}(U_{DEV})$, and $NM_{TL}(I_{DEV})$ sets.

3. **Check Preconditions for Statistical Testing:** We separately check for each of the above mentioned sets if the values within these sets follow a normal distribution using the Shapiro-Wilk test (Section 2.3.5.1). Moreover, we separately test for homoscedasticity using the Brown-Forsythe test (Section 2.3.5.2) between $NM_C(U_{DEV})$ and $NM_C(I_{DEV})$, as well as between $NM_{TL}(U_{DEV})$ and $NM_{TL}(I_{DEV})$. These tests are performed in order to choose the correct statistical significance test in the next step.

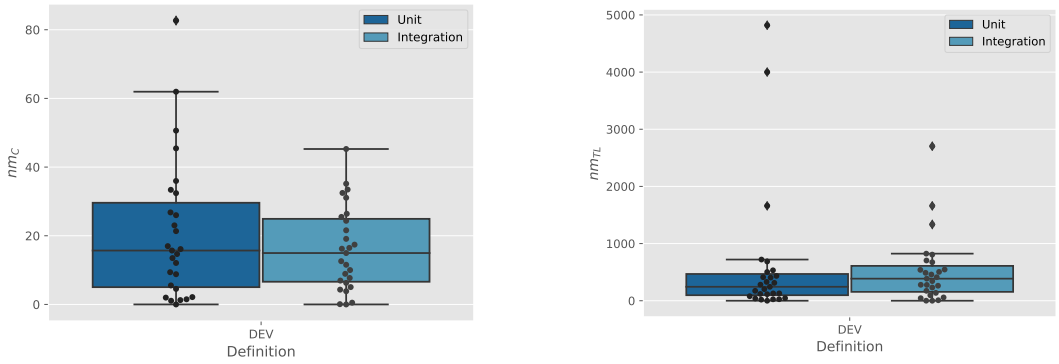


Figure 5.2.: Box-plots of the nm_C metric (left) and nm_{TL} metric (right) for unit and integration tests and the DEV rule set. The points in the plot represent the concrete values for each project.

4. **Statistical Testing:** Based on the results of the previous step, we chose a fitting one-sided significance test (Section 2.3.1) to test for differences between $NM_C(U_{DEV})$ and $NM_C(I_{DEV})$, as well as between $NM_{TL}(U_{DEV})$ and $NM_{TL}(I_{DEV})$ due to the same reasons as explained in Section 5.2.1.

5.3.2. Results

Table 5.5 depicts the nm_C and nm_{TL} metrics for each project for the DEV rule set. Overall, there are 14 projects for the nm_C metric, where the trend of developing more integration than unit tests holds true. These projects are mostly Java projects (11), while only three Python projects follow the trend. The numbers are similar for the nm_{TL} metric: overall, there is one project more than for the nm_C metric, where the trend of developing more integration than unit tests holds true. These 15 projects include the same 11 Java projects as for the nm_C metric. Furthermore, one more Python project is part of this set. Nevertheless, the mean for both metrics is higher for unit (20.46 (nm_C), 608.93 (nm_{TL})) than for the integration tests (16.19 (nm_C), 516.83 (nm_{TL})), while all sets have a high standard deviation. This gives a first hint that there are more unit than integration tests if we rely on the developer classification.

Figure 5.2 depicts box-plots of the nm_C (left) and the nm_{TL} (right) metrics. If we compare them visually, we can determine that the shown median for both sets (i.e., unit and integration tests) are similar in both figures. This indicates that there is no difference in both metrics between unit and integration tests.

While we do not see a difference in the data (Table 5.5) or visually (Figure 5.2) we need to assess it statistically. Hence, we performed several statistical tests. The concrete p-values and test statistics for each statistical test is reported in Appendix C. The Shapiro-Wilk tests showed that only the nm_C values for the set of integration tests follow a normal distribution,

Project	$nm_C(U_{DEV})$	$nm_C(I_{DEV})$	$nm_{TL}(U_{DEV})$	$nm_{TL}(I_{DEV})$
commons-beanutils	0.00	35.16	0.00	1659.91
commons-codec	35.96	6.94	412.21	95.86
commons-collections	82.67	14.97	4000.03	1334.89
commons-io	14.68	24.36	281.17	807.20
commons-lang	32.41	19.10	502.65	262.77
commons-math	26.00	5.05	686.90	137.23
druid	1.26	16.22	26.52	279.01
fastjson	1.99	26.40	19.23	276.86
google-gson	13.51	33.46	128.79	488.47
guice	5.53	17.41	114.01	343.58
HikariCP	1.50	8.91	39.17	504.85
jackson-core	2.14	16.47	44.73	546.01
jfreechart	15.70	0.51	243.96	8.67
joda-time	16.12	32.47	312.89	704.04
jsoup	21.35	10.01	434.25	231.29
mybatis-3	8.80	12.60	203.29	411.57
zxing	1.07	11.55	25.05	458.80
ChatterBot	12.08	25.49	79.26	386.35
csvkit	61.95	0.00	720.69	0.00
dpark	4.53	0.08	119.12	0.95
mrjob	45.45	4.31	4818.83	2702.85
networkx	23.03	21.59	532.44	540.06
pyramid	50.62	7.70	1660.72	173.30
python-telegram-bot	26.80	6.28	329.06	44.42
rq	33.38	3.85	404.73	57.77
schematics	9.38	31.07	128.09	673.68
scrapy	16.98	45.27	173.44	823.94
Mean	20.92	16.19	608.93	516.83
StDev	20.46	12.18	1152.01	587.75

Table 5.5.: Normalized test count values (nm_C) and normalized tl values (nm_{TL}) for each project.

while all other tested sets do not. The Brown-Forsythe test showed, that all tested sets are homoscedastic. The U-statistic was not significant at the .005 critical alpha level, for all tested sets. Therefore, we fail to reject H_0 for all of the tested sets.

In addition to the one-sided Mann-Whitney-U test, we performed a two-sided test to assess, if there are any differences at all between unit and integration tests for the two assessed metrics. The U-statistic was not significant at the .005 critical alpha level for all tested sets. Hence, in addition to the one-sided test we also fail to reject the H_0 for all tested sets for the two-sided test. Therefore, we conclude that there is no statistically significant difference between unit and integration tests according to the developer classification.

Answer to RQ 1.2: Table 5.5 and Figure 5.2 show that there are no differences in the median and the mean between the amount of unit and integration tests (measured in nm_C and nm_{TL}), while a high standard deviation exists for the nm_{TL} metric. To assess if the table and figures provide a correct view, we performed several statistical tests, which all failed to reject H_0. Also the two-sided test failed to reject H_0. Therefore, we conclude that when relying on the developer classification, the trend of developing more integration than unit tests is not visible in open-source software projects. Furthermore, our two-sided test shows that there is no difference at all between unit and integration tests for the examined metrics according to the developer classification.

5.4. Evaluation of RQ 1.3: Developer Classification according to Definitions

Within this RQ we want to compare the classification of tests by definition (RQ 1.1) with the classification done by developers (RQ 1.2). The answer to this RQ provides us with a deeper understanding of the current practice of testing in the real world and the (potential) difference of testing between academia/education and practice. This section presents our analysis procedure (Section 5.4.1), together with its results (Section 5.4.2).

5.4.1. Analysis Procedure

Within our analysis, we compare the number of tests and their tl of the U_{DEV} set with the U_{IEEE} and U_{ISTQB} sets, as well as the I_{DEV} set with the I_{IEEE} and I_{ISTQB} sets. However, as we believe that there is a connection between the unit tests sets, as well as the integration test sets, we can not perform the same kind of analysis as done in sections 5.2.1 and 5.3.1. The significance tests used in these sections have the precondition that the tested samples are independent of each other, which is violated if we compare, e.g., the U_{DEV} and U_{ISTQB} sets, due to their relationship. Hence, instead of statistically testing for differences, we perform different set operations to gather the overlap and differences between the developer classification and the classification by the definitions. Additionally, we calculate the tl of the test sets, which are newly created by the set operations.

Our analysis process is described in detail in the following:

1. **Gather the Data:** We gather the tests with their classification results that were created via the DEV, IEEE, ISTQB rule sets for all subject projects. Furthermore, we query the *tl* for all test sets, i.e., all $tl(X)$ with $X \in \{U_{DEV}, I_{DEV}, U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}\}$.

2. **Set Operations:** As we want to assess if the developers of the projects classify their tests according to the definitions, we perform four set operations for each definition. To gather the number of tests that are classified **according to the definitions**, we calculate the intersection between the unit and integration tests as classified by the developers and by the definitions, i.e., $U_{DEV} \cap U_{IEEE}, U_{DEV} \cap U_{ISTQB}, I_{DEV} \cap I_{IEEE}, I_{DEV} \cap I_{ISTQB}$. Furthermore, as we also want to assess the number of tests that are **missclassified** according to the definitions, we calculate the difference between the unit and integration tests as classified by the developers and by the definitions, i.e., $U_{DEV} \setminus U_{IEEE}, U_{DEV} \setminus U_{ISTQB}, I_{DEV} \setminus I_{IEEE}, I_{DEV} \setminus I_{ISTQB}$. In a last step, we sum up the number of *tl* for each of these newly created sets.

3. **Compare Results:** Finally, we compare the number of tests within the newly created sets visually and by value to determine if there is a difference and assess its magnitude.

5.4.2. Results

Tables 5.6 and 5.7 depict the number of tests and their *tl* within the sets that were created by the different set operations described in Section 5.4.1. Table 5.6 shows the comparison of tests based on the IEEE definition, while Table 5.7 the comparison of tests based on the ISTQB definition. However, both tables show a similar trend. They depict that on average more test get classified correctly according to the definitions (1037.59 (IEEE), 972.37 (ISTQB)), than they get missclassified according to them (781.63 (IEEE), 846.85 (ISTQB)). On average, most misclassifications occur for unit tests, as both of the tables highlight (636.19 (IEEE), 793.74 (ISTQB)). The results are changing, if we compare the *tl* of the tests that are contained with the sets. We can see that on average more *tl* get missclassified (27452.04 (IEEE), 29508.89 (ISTQB)) than correctly classified (27174.22 (IEEE), 25117.37 (ISTQB)) according to the definitions. However, the differences between these values are rather low. Furthermore, both tables depict that the standard deviation is rather high for the shown data, indicating a large variance between projects. This is supported by the raw numbers shown in the tables. For example, *HikariCP* has substantially more tests that are classified according to the definitions (98 (IEEE), 97 (ISTQB)) than missclassified ones (20 (IEEE), 21 (ISTQB)), while it is vice versa for other projects like *commons-math*, where less tests are classified according to the definitions (1670 (IEEE), 1283 (ISTQB)) and more are misclassified (4814 (IEEE), 5201 (ISTQB)).

Project	$\lvert U_{DEV} \cap U_{IEEE} \rvert (tl)$		$\lvert I_{DEV} \cap I_{IEEE} \rvert (tl)$		$\lvert U_{DEV} \setminus U_{IEEE} \rvert (tl)$		$\lvert I_{DEV} \setminus I_{IEEE} \rvert (tl)$	
commons-beanutils	0	(0)	890	(44510)	0	(0)	285	(10966)
commons-codec	572	(6451)	3	(21)	143	(1745)	135	(1885)
commons-collections	1777	(61310)	761	(79334)	3244	(181624)	148	(1738)
commons-io	259	(3008)	335	(9193)	169	(5188)	375	(14337)
commons-lang	2068	(28730)	36	(754)	435	(10087)	1439	(19538)
commons-math	695	(10415)	975	(27065)	4734	(133038)	80	(1594)
druid	47	(376)	3786	(65707)	251	(5890)	47	(209)
fastjson	58	(357)	3609	(38927)	233	(2459)	257	(1616)
google-gson	185	(1857)	691	(10197)	106	(918)	30	(328)
guice	9	(48)	513	(10279)	160	(3435)	19	(217)
HikariCP	9	(110)	89	(5514)	8	(334)	12	(208)
jackson-core	29	(249)	667	(22474)	60	(1612)	18	(241)
jfreechart	224	(2604)	65	(1125)	1881	(30115)	4	(38)
joda-time	161	(2520)	2758	(59895)	1217	(24227)	18	(290)
jsoup	59	(635)	183	(4274)	342	(7521)	5	(70)
mybatis-3	215	(1423)	608	(20130)	216	(8533)	9	(26)
zxing	6	(101)	265	(13114)	28	(695)	102	(1463)
ChatterBot	36	(237)	169	(2798)	55	(360)	23	(112)
csvkit	36	(345)	0	(0)	141	(1714)	0	(0)
dpark	17	(209)	0	(0)	40	(1289)	1	(12)
mrjob	906	(15214)	153	(99497)	771	(162601)	6	(238)
networkx	272	(6484)	786	(21744)	1107	(25404)	507	(10600)
pyramid	1147	(21524)	275	(6373)	1054	(50684)	60	(1162)
python-telegram-bot	181	(1675)	62	(505)	327	(4562)	57	(337)
rq	44	(360)	20	(302)	138	(1847)	1	(13)
schematics	26	(195)	209	(4999)	48	(815)	36	(313)
scrapy	165	(379)	904	(18157)	269	(4054)	253	(2903)
Mean	340.85	(6178.37)	696.74	(20995.85)	636.19	(24842.63)	145.44	(2609.41)
StDev	541.27	(13115.32)	1029.85	(26838.70)	1087.38	(50204.99)	289.47	(5039.72)

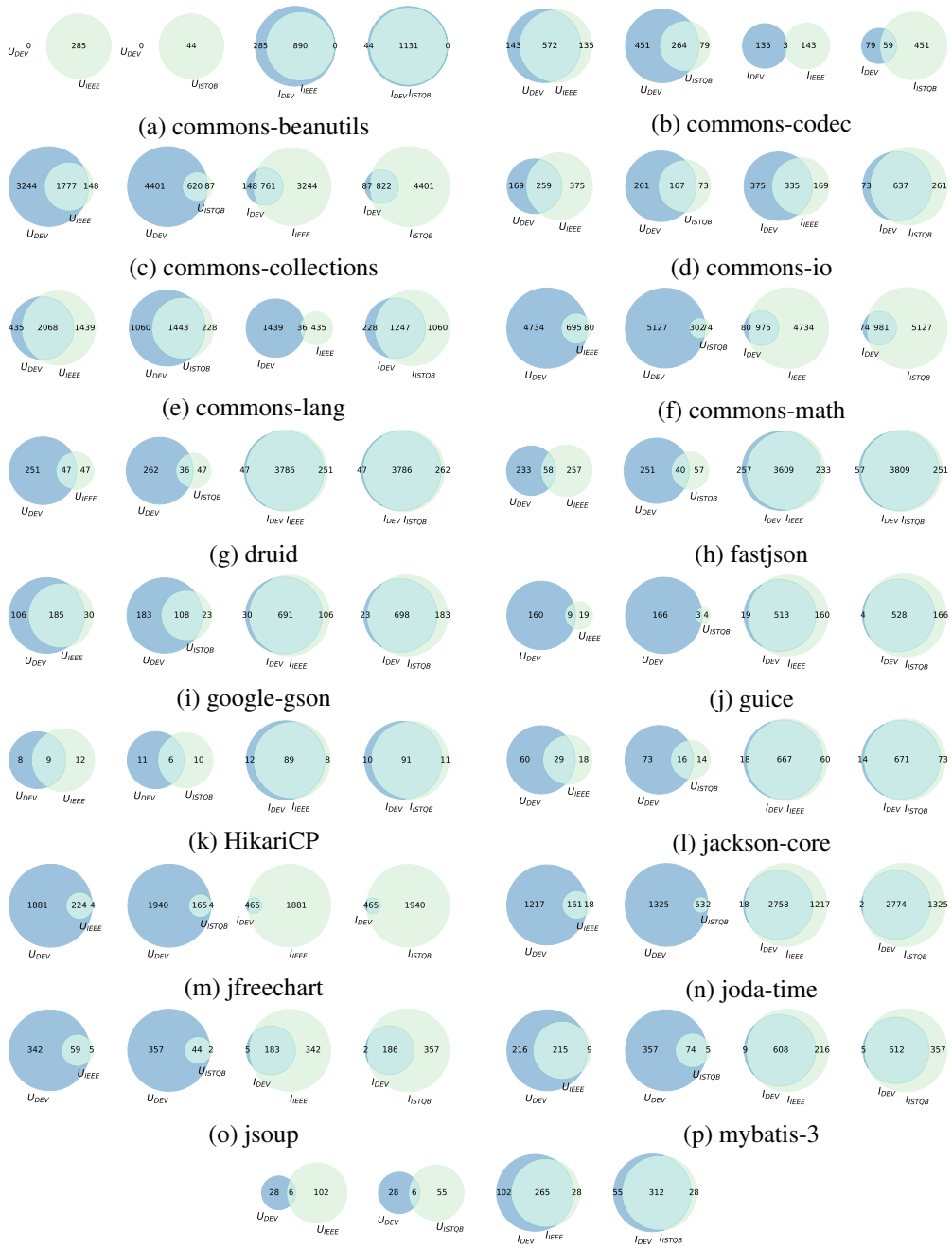
Table 5.6.: Number of tests and their tl within the sets created by different set operations. The sets created by intersections contain tests that were classified by the developers **according to the IEEE definition**. The sets created by differencing contain tests that are **missclassified** according to the IEEE definition.

Project	$\lvert U_{DEV} \cap U_{ISTQB} \rvert (tl)$		$\lvert I_{DEV} \cap I_{ISTQB} \rvert (tl)$		$\lvert U_{DEV} \setminus U_{ISTQB} \rvert (tl)$		$\lvert I_{DEV} \setminus I_{ISTQB} \rvert (tl)$	
commons-beanutils	0	(0)	1131	(54538)	0	(0)	44	(938)
commons-codec	264	(2355)	59	(457)	451	(5841)	79	(1449)
commons-collections	620	(15009)	822	(80314)	4401	(227925)	87	(758)
commons-io	167	(1917)	637	(22123)	261	(6279)	73	(1407)
commons-lang	1443	(16188)	1247	(17168)	1060	(22629)	228	(3124)
commons-math	302	(3430)	981	(27298)	5127	(140023)	74	(1361)
druid	36	(246)	3786	(65707)	262	(6020)	47	(209)
fastjson	40	(215)	3809	(40137)	251	(2601)	57	(406)
google-gson	108	(1074)	698	(10301)	183	(1701)	23	(224)
guice	3	(13)	528	(10452)	166	(3470)	4	(44)
HikariCP	6	(73)	91	(5648)	11	(371)	10	(74)
jackson-core	16	(130)	671	(22517)	73	(1731)	14	(198)
jfreechart	165	(1936)	65	(1125)	1940	(30783)	4	(38)
joda-time	53	(471)	2774	(60177)	1325	(26276)	2	(8)
jsoup	44	(457)	186	(4316)	357	(7699)	2	(28)
mybatis-3	74	(506)	612	(20144)	357	(9450)	5	(12)
zxing	6	(101)	312	(13965)	28	(695)	55	(612)
ChatterBot	36	(237)	185	(2874)	55	(360)	7	(36)
csvkit	29	(285)	0	(0)	148	(1774)	0	(0)
dpark	13	(146)	1	(12)	44	(1352)	0	(0)
mrjob	466	(3970)	155	(99604)	1211	(173845)	4	(131)
networkx	91	(1691)	1027	(27301)	1288	(30197)	266	(5043)
pyramid	662	(8815)	305	(7099)	1539	(63393)	30	(436)
python-telegram-bot	125	(896)	64	(513)	383	(5341)	55	(329)
rq	8	(46)	21	(315)	174	(2161)	0	(0)
schematics	7	(106)	216	(5025)	67	(904)	29	(287)
scrapy	165	(379)	922	(18347)	269	(4054)	235	(2713)
Mean	183.30	(2247.85)	789.07	(22869.52)	793.74	(28773.15)	53.11	(735.74)
StDev	310.68	(4274.48)	1046.06	(26922.04)	1267.75	(57794.41)	74.05	(1183.27)

Table 5.7.: Number of tests and their tl within the sets created by different set operations. The sets created by intersections contain tests that were classified by the developers **according to the ISTQB definition**. The sets created by differencing contain tests that are **missclassified** according to the ISTQB definition.

These results are also supported by the Venn-diagrams presented in figures 5.3 and 5.4. Figure 5.3 depicts Venn-diagrams that show the number of tests and their overlap between U_{DEV} and U_{IEEE}, U_{DEV} and U_{ISTQB}, I_{DEV} and I_{IEEE}, I_{DEV} and I_{ISTQB} for all Java projects. Figure 5.4 shows the same data for all Python projects.

Answer to RQ 1.3: Tables 5.6 and 5.7 highlight that while on average more tests get classified according to the definitions than misclassified, there is still a large gap between the developer classification of tests and the classification based on the definitions. This gap is larger for tests that get classified as unit tests by the developers, showing that especially the current definition of unit tests might not fit to modern development practices. This is supported by the Venn-diagrams shown in figures 5.3 and 5.4.



(a) commons-beanutils (b) commons-codec

(c) commons-collections (d) commons-io

(e) commons-lang (f) commons-math

(g) druid (h) fastjson

(i) google-gson (j) guice

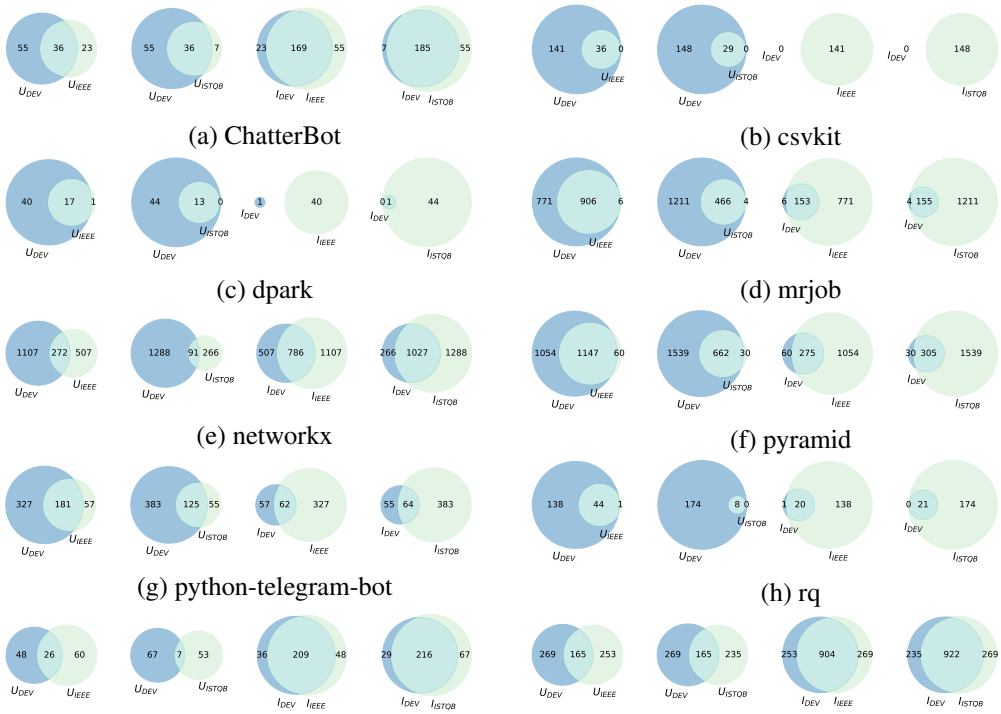
(k) HikariCP (l) jackson-core

(m) jfreechart (n) joda-time

(o) jsoup (p) mybatis-3

(q) zxing

Figure 5.3.: Venn-diagrams showing the number of tests and their overlap between U_{DEV} and U_{IEEE}, U_{DEV} and U_{ISTQB}, I_{DEV} and I_{IEEE}, I_{DEV} and I_{ISTQB} for all Java projects.



(a) ChatterBot (b) csvkit

(c) dpark (d) mrjob

(e) networkx (f) pyramid

(g) python-telegram-bot (h) rq

(i) schematics (j) scrapy

Figure 5.4.: Venn-diagrams showing the number of tests and their overlap between U_{DEV} and U_{IEEE}, U_{DEV} and U_{ISTQB}, I_{DEV} and I_{IEEE}, I_{DEV} and I_{ISTQB} for all Python projects.

6. Quantitative Evaluation of the Differences between Unit and Integration Tests

This section describes the analysis procedures and results of the quantitative evaluation of the differences between unit and integration tests. The quantitative evaluation was done by extending the case study that produced the results highlighted in Section 5. For each of the RQs that we evaluate quantitatively, we include a section (sections 6.1 - 6.3) in which the used data set, the analysis procedure, and the results are described. Based on the results of RQ 1.3, where we observed that developers do not classify their tests according to any of the definitions, we decided to perform all subsequent analyses with both definitions and the developer classification.

6.1. Evaluation of RQ 2.1: Test Execution Time

This RQ is concerned with the difference in the test execution time between unit and integration tests. We want to assess if the difference stated in the standard literature (i.e., unit tests are faster than integration tests) is reflected in the data collected from current open-source software projects. Hence, we first describe the data that we used within our analysis in Section 6.1.1. Afterwards, we specify our analysis procedure (Section 6.1.2) and report the results of our analysis (Section 6.1.3).

6.1.1. Data set Description

For the analysis of this RQ, we reuse the data set described in Section 5.1. Additionally, we define $exe_{SUM}(X) := \sum_{x \in X} exeTime(x)$ as the average execution time for all tests within the test set X and $pl(X) := \sum_{x \in X} pLOC(x)$ as the sum of all production lines that are covered by tests in X. The collection of the execution time is detailed in Section 4.2.9 and the collection of the pLOC in Section 4.2.5.

Table 6.1 shows the accumulated execution time measured in ms for each project and each test set. This table depicts that the integration tests have a higher execution time (70.91 ms (IEEE), 75.67 ms (ISTQB), 45.76 ms (DEV)) than unit tests (9.99 ms (IEEE), 5.23 ms (ISTQB), 35.15 ms (DEV)), while also having a higher standard deviation (102.14 (IEEE), 109.23 (ISTQB), 77.99 (DEV) in contrast to 23.40 (IEEE), 20.81 (ISTQB), 61.66 (DEV)). As the standard deviation highlights, there are some projects with a rather high overall execution time (e.g., *scrapy*), while others have a low execution time (e.g., *pyramid*).

Project	$exe_{SUM}(U_{IEEE})$	$exe_{SUM}(I_{IEEE})$	$exe_{SUM}(U_{ISTQB})$	$exe_{SUM}(I_{ISTQB})$	$exe_{SUM}(U_{DEV})$	$exe_{SUM}(I_{DEV})$
commons-beanutils	4.55	11.81	0.08	16.28	0.00	16.36
commons-codec	11.30	0.88	3.80	8.38	8.10	4.08
commons-collections	12.10	3.22	0.36	14.96	14.74	0.58
commons-io	16.02	60.45	5.31	71.15	36.91	39.56
commons-lang	25.42	4.00	5.79	23.62	14.79	14.63
commons-math	1.04	201.84	0.31	202.57	159.39	43.49
druid	0.73	73.75	0.25	74.22	6.58	67.89
fastjson	0.35	44.28	0.28	44.35	1.17	43.46
google-gson	0.15	1.20	0.08	1.27	0.24	1.12
guice	0.10	9.48	0.01	9.56	2.71	6.86
HikariCP	0.09	101.16	0.03	101.22	6.35	94.90
jackson-core	0.71	11.22	0.68	11.25	0.22	11.71
jfreechart	0.23	5.09	0.19	5.12	4.15	1.17
joda-time	0.08	7.69	0.00	7.77	4.15	3.62
jsoup	0.11	6.59	0.08	6.62	2.66	4.04
mybatis-3	2.51	73.53	2.09	73.95	41.47	34.57
zxing	7.53	292.60	0.08	300.05	6.82	293.32
ChatterBot	0.43	66.65	0.05	67.02	0.50	66.57
csvkit	0.00	15.36	0.00	15.36	15.36	0.00
dpark	2.87	72.89	2.86	72.90	75.75	0.00
mrjob	60.21	388.41	1.03	447.59	263.65	184.97
networkx	2.13	48.85	1.73	49.26	34.63	16.35
pyramid	0.25	13.89	0.11	14.04	8.90	5.24
python-telegram-bot	11.73	99.26	7.18	103.81	96.05	14.94
rq	0.11	17.38	0.01	17.47	13.86	3.63
schematics	0.07	0.54	0.04	0.57	0.11	0.50
scrapy	109.05	282.60	108.85	282.80	129.80	261.85
Mean	9.99	70.91	5.23	75.67	35.15	45.76
StDev	23.40	102.14	20.81	109.23	61.66	77.99

Table 6.1.: Accumulated execution time (in ms) of each project for each test set.

Table 6.2 depicts the number of production lines covered by the tests within the different test sets. The values in this table are given in pKLOC. The numbers in brackets depict the *pl* per test in the test set. This table highlights that (in the mean) integration tests cover more code (906.32 (IEEE), 902.78 (ISTQB), 427.74 (DEV)) than unit tests (26.09 (IEEE), 3.94 (ISTQB), 195.61 (DEV)). This also holds true if we compare the *pl* values per test in the test set (0.04 (IEEE), 0.02 (ISTQB), 0.26 (DEV)) than integration tests (0.44 (IEEE), 0.40 (ISTQB), 0.47 (DEV)). Furthermore, these numbers have a lower standard deviation than the absolute numbers for all test sets.

6.1.2. Analysis Procedure

We assess the difference in the execution time by evaluating the execution time per covered line of code for unit and integration tests. We decided against an analysis of the raw execution time (i.e., execution time without normalization), as there would be a bias in the data: integration tests cover more code and are therefore (mostly) slower than unit tests. Therefore, we can assess which testing techniques are faster per covered production line with our results. The analysis process is similar to the one presented in Section 5.2.1. Nevertheless, different data is used and therefore we explain each step of the analysis in detail in the following.

1. **Gather the Data:** We gather the $exe_{SUM}(X)$ and $pl(X)$ for each $X \in \{U_{IEEE}, U_{ISTQB}, I_{IEEE}, I_{ISTQB}, U_{DEV}, I_{DEV}\}$ for all projects.

2. **Calculate Ratio:** As we want to assess, if there are differences in the execution time per covered production line of unit and integration tests, we need to calculate this ratio first. Hence, we define $rat_{EXE}(X,p) := \frac{exe_{SUM}(X(p))}{pl(X(p))}$ as the ratio of the accumulated execution time to the covered production lines of one test set X for one project p. Furthermore, we define the set of all rat_{EXE} ratios of all projects for a test set X as $RAT_{EXE}(X) := \{rat_{EXE}(X,p) | p \in Projects\}$.

3. **Check Preconditions for Statistical Testing:** We separately check for each of the $RAT_{EXE}(X)$ if the ratios inside these sets follow a normal distribution using the Shapiro-Wilk test (Section 2.3.5.1). Moreover, we separately test for homoscedasticity using the Brown-Forsythe test (Section 2.3.5.2) between $RAT_{EXE}(U_{IEEE})$ and $RAT_{EXE}(I_{IEEE})$, $RAT_{EXE}(U_{ISTQB})$ and $RAT_{EXE}(I_{ISTQB})$, as well as between $RAT_{EXE}(U_{DEV})$ and $RAT_{EXE}(I_{DEV})$. These tests are done to be able to choose the correct statistical significance test in the next step.

4. **Statistical Testing:** Based on the results of the previous step, we chose a fitting two-sided significance test (Section 2.3.1) to test for differences between the values within $RAT_{EXE}(U_{IEEE})$ and $RAT_{EXE}(I_{IEEE})$, $RAT_{EXE}(U_{ISTQB})$ and $RAT_{EXE}(I_{ISTQB})$, as well as $RAT_{EXE}(U_{DEV})$ and $RAT_{EXE}(I_{DEV})$. We choose a two-sided test here as we want to

Project	$pl(U_{IEEE})$	$pl(I_{IEEE})$	$pl(U_{ISTQB})$	$pl(I_{ISTQB})$	$pl(U_{DEV})$	$pl(I_{DEV})$
commons-beanutils	19.23 (0.07)	212.68 (0.24)	0.56 (0.01)	231.35 (0.20)	0.00 (0.00)	231.91 (0.20)
commons-codec	34.15 (0.05)	5.38 (0.04)	7.72 (0.02)	31.82 (0.06)	34.77 (0.05)	4.77 (0.03)
commons-collections	93.15 (0.05)	510.68 (0.13)	8.03 (0.01)	595.80 (0.11)	332.17 (0.07)	271.66 (0.30)
commons-io	22.95 (0.04)	36.21 (0.07)	4.46 (0.02)	54.70 (0.06)	16.71 (0.04)	42.45 (0.06)
commons-lang	272.34 (0.08)	48.75 (0.10)	21.02 (0.01)	300.08 (0.13)	99.81 (0.04)	221.29 (0.15)
commons-math	30.18 (0.04)	1480.22 (0.26)	5.38 (0.01)	1505.01 (0.25)	1238.93 (0.23)	271.47 (0.26)
druid	1.99 (0.02)	4039.12 (1.00)	1.34 (0.02)	4039.77 (1.00)	295.84 (0.99)	3745.27 (0.98)
fastjson	15.72 (0.05)	2718.42 (0.71)	2.74 (0.03)	2731.41 (0.67)	88.20 (0.30)	2645.95 (0.68)
google-gson	21.02 (0.10)	314.71 (0.39)	12.73 (0.10)	323.00 (0.37)	32.08 (0.11)	303.64 (0.42)
guice	0.52 (0.02)	774.93 (1.15)	0.06 (0.01)	775.39 (1.12)	124.53 (0.74)	650.92 (1.22)
HikariCP	0.36 (0.02)	41.62 (0.43)	0.25 (0.02)	41.73 (0.41)	3.54 (0.21)	38.44 (0.38)
jackson-core	1.73 (0.04)	327.80 (0.45)	0.96 (0.03)	328.58 (0.44)	20.80 (0.23)	308.73 (0.45)
jfreechart	3.80 (0.02)	622.18 (0.32)	2.21 (0.01)	623.77 (0.31)	534.80 (0.25)	91.19 (1.32)
joda-time	6.01 (0.03)	869.46 (0.22)	0.33 (0.01)	875.14 (0.21)	254.99 (0.19)	620.48 (0.22)
jsoup	1.89 (0.03)	417.09 (0.79)	1.29 (0.03)	417.69 (0.77)	255.12 (0.64)	163.86 (0.87)
mybatis-3	5.21 (0.02)	939.97 (1.14)	1.81 (0.02)	943.38 (0.97)	189.60 (0.44)	755.59 (1.22)
zxing	7.20 (0.07)	218.14 (0.74)	1.44 (0.02)	223.90 (0.66)	12.97 (0.38)	212.36 (0.58)
ChatterBot	1.14 (0.02)	29.31 (0.13)	0.75 (0.02)	29.70 (0.12)	4.18 (0.05)	26.28 (0.14)
csvkit	0.70 (0.02)	19.76 (0.14)	0.56 (0.02)	19.90 (0.13)	20.47 (0.12)	0.00 (0.00)
dpark	1.33 (0.07)	35.90 (0.90)	0.86 (0.07)	36.36 (0.81)	37.22 (0.65)	0.01 (0.01)
mrjob	88.74 (0.10)	706.36 (0.76)	5.03 (0.01)	790.07 (0.58)	549.93 (0.33)	245.17 (1.54)
networkx	25.66 (0.03)	362.07 (0.19)	9.69 (0.03)	378.04 (0.16)	274.87 (0.20)	112.86 (0.09)
pyramid	35.30 (0.03)	1013.81 (0.76)	9.79 (0.01)	1039.32 (0.56)	730.67 (0.33)	318.44 (0.95)
python-telegram-bot	4.94 (0.02)	38.12 (0.10)	3.02 (0.02)	40.04 (0.09)	35.31 (0.07)	7.74 (0.07)
rq	2.87 (0.06)	35.42 (0.22)	0.04 (0.01)	38.25 (0.20)	35.19 (0.19)	3.10 (0.15)
schematics	2.27 (0.03)	64.59 (0.25)	0.83 (0.01)	66.03 (0.23)	12.71 (0.17)	53.84 (0.22)
scrapy	3.96 (0.01)	243.51 (0.21)	3.50 (0.01)	243.97 (0.20)	45.96 (0.11)	201.52 (0.17)
Mean	26.09 (0.04)	597.27 (0.44)	3.94 (0.02)	619.41 (0.40)	195.61 (0.26)	427.74 (0.47)
StDev	54.81 (0.02)	906.32 (0.35)	4.88 (0.02)	902.78 (0.32)	284.65 (0.24)	836.84 (0.46)

Table 6.2.: *pl* of each project for each test set. The numbers in the brackets depict the pKLOC per test.

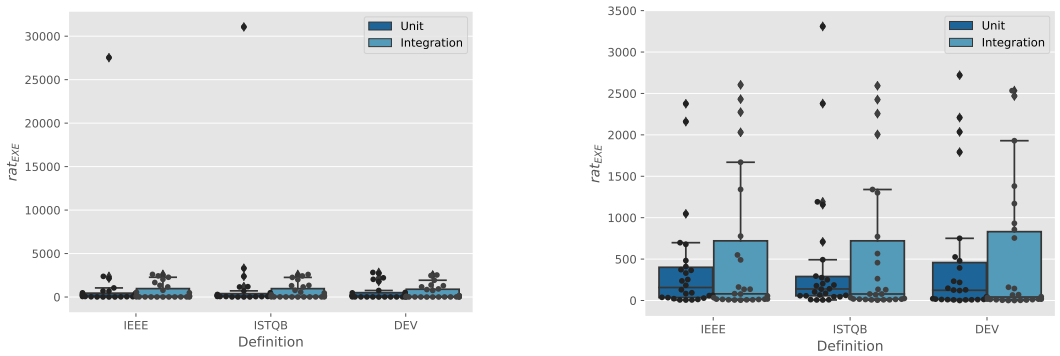


Figure 6.1.: Box-plot of the rat_{EXE} ratio for unit and integration tests and the IEEE and ISTQB definitions, as well as the DEV classification. The right box-plot is a zoomed-in version of the left box-plot. The points in the plot represent the concrete values for each project.

assess if there are any statistically significant differences at all. All concrete p-values and test statistics for these tests are presented in Appendix C.2.

6.1.3. Results

Table 6.3 depicts that in the mean unit tests have a higher accumulated execution time per pKLOC (1396.83 (IEEE), 1575.05 (ISTQB), 551.08 (DEV)) than integration tests (597.17 (IEEE), 585.03 (ISTQB), 517.46 (DEV)). Nevertheless, the unit test sets contain a higher standard deviation (5258.70 (IEEE), 5943.81 (ISTQB), 894.58 (DEV)) than the integration test sets (866.36 (IEEE), 844.83 (ISTQB), 789.52 (DEV)). This highlights that the common wisdom that unit tests are faster (with respect to their execution time) cannot be supported by our data. In fact, the data shows us the opposite: integration tests are faster per covered line of code than unit tests.

The box-plot in Figure 6.1 shows the distribution of the values from Table 6.3. The box-plot also shows that we have a wide range of values, as also highlighted by the standard deviation in Table 6.3. Nevertheless, Figure 6.1 shows that most of the values are in the lower part of the figure (i.e., between 0 and 500). Furthermore, the similar shape of the boxes highlight that there are no large differences between unit and integration tests with respect to their execution time per covered production line, neither for both definitions nor for the developer classification.

As explained in Section 6.1.2, we additionally performed several statistical tests. These tests showed, that none of the tested sets follow a normal distribution. The Brown-Forsythe test showed, that all tested sets are homoscedastic. The U-statistic was not significant at the .005 critical alpha level for neither of the tested sets. Therefore, we fail to reject H_0 for all of the tested sets and can conclude that there are no statistically significant differences in

Project	U_{IEEE}	I_{IEEE}	U_{ISTQB}	I_{ISTQB}	U_{DEV}	I_{DEV}
commons-beanutils	236.62	55.52	141.07	70.36	0.00	70.54
commons-codec	330.99	162.64	492.35	263.39	232.87	856.69
commons-collections	129.94	6.30	45.34	25.10	44.38	2.14
commons-io	698.02	1669.28	1191.44	1300.81	2208.69	931.94
commons-lang	93.32	82.09	275.74	78.72	148.15	66.12
commons-math	34.43	136.36	57.59	134.60	128.65	160.22
druid	364.46	18.26	186.90	18.37	22.25	18.13
fastjson	21.94	16.29	103.03	16.24	13.30	16.42
google-gson	7.19	3.82	6.44	3.94	7.36	3.68
guice	183.91	12.23	250.00	12.33	21.80	10.54
HikariCP	253.44	2430.79	137.10	2425.48	1791.54	2469.18
jackson-core	410.27	34.22	708.46	34.24	10.57	37.92
jfreechart	61.25	8.17	87.70	8.22	7.77	12.79
joda-time	12.80	8.85	9.09	8.87	16.28	5.83
jsoup	58.73	15.79	62.02	15.84	10.42	24.65
mybatis-3	482.26	78.22	1157.72	78.39	218.74	45.75
zxing	1046.84	1341.38	56.41	1340.15	525.60	1381.21
ChatterBot	372.38	2273.87	66.76	2256.29	120.42	2533.39
csvkit	5.68	777.25	7.10	771.74	750.71	0.00
dpark	2159.76	2030.36	3309.39	2004.73	2035.47	0.00
mrjob	678.45	549.88	204.50	566.52	479.43	754.45
networkx	83.18	134.93	178.23	130.30	126.00	144.90
pyramid	7.00	13.70	10.83	13.50	12.18	16.46
python-telegram-bot	2376.05	2603.94	2377.17	2592.95	2719.96	1929.42
rq	38.49	490.48	294.29	456.82	393.77	1170.19
schematics	28.97	8.37	45.10	8.61	8.29	9.30
scrapy	27538.14	1160.53	31064.67	1159.18	2824.45	1299.42
Mean	1396.83	597.17	1575.05	585.03	551.08	517.46
StDev	5258.70	866.36	5943.81	844.83	894.58	789.52

Table 6.3.: rat_{EXE} of each project for each test set.

the execution time per covered production line between unit and integration tests, neither for both tested definitions nor for the developer classification.

Answer to RQ 2.1: Table 6.3 and Figure 6.1 highlight that there are no differences in the execution time per covered production line between unit and integration tests. In fact, Table 6.3 shows that (in the mean) the execution time per covered production line is higher for unit tests than for integration tests. In addition, all statistical tests failed to reject H_0. Overall, we can conclude that there is no statistically significant difference between unit and integration tests in terms of their execution time per covered production line of code.

Project	#Analyzed Tests	#Unique Mutants
commons-beanutils	1175	11310
commons-codec	853	9059
commons-collections	5930	26464
commons-io	1138	9593
commons-lang	3978	36549
commons-math	6484	113469
druid	4127	123835
fastjson	4147	48289
google-gson	1012	8816
guice	688	10851
HikariCP	117	4967
jackson-core	774	34361
jfreechart	2174	96104
joda-time	4153	32951
jsoup	588	14171
mybatis-3	1043	17126
zxing	401	29592
Overall	38782	627507

Table 6.4.: Number of analyzed tests and unique mutants for each project.

6.2. Evaluation of RQ 2.2: Test Effectiveness

This RQ is concerned with the difference in the test effectiveness between unit and integration tests. We divided this question into two different parts: first, we analyze if there is an overall difference in the test effectiveness between unit and integration tests. Second, we analyze if there is a difference in the test effectiveness between unit and integration tests per defect type. This analysis is done, as the standard literature agrees, that integration tests are mostly detecting integration defects, while unit tests detect other kind of defects. Within this RQ, we want to evaluate if this difference is reflected in actual open-source projects. Therefore, we describe the data that we have used to analyze this RQ (Section 6.2.1), our analysis methodology for both sub questions (Section 6.2.2), as well as the results of both analyses (Section 6.2.3).

6.2.1. Data set Description

To evaluate this RQ, we make use of mutation testing (Section 2.2). Hence, we integrate mutants into the source code of the projects to assess the defect detection capabilities of their tests, according to the procedure explained in Section 4.2.6. Table 6.4 shows the number of

unique mutants[10] that are generated and the number of analyzed tests for each project. As explained in Section 4.2.6 we were only able to gather the mutation detection capabilities for Java projects, as a functioning and fitting Python mutation testing framework was not available. The number of analyzed tests shown in Table 6.4 can be lower than the overall number of tests for some projects (e.g., *commons-beanutils* or *commons-collections*), as our mutation testing tool might not be able to run the tests in isolation, e.g., if tests fail when they are executed alone as they depend on other tests that must be run beforehand).

We create two different data sets for the analysis of this RQ. These data sets represent the different perspectives on the RQ at hand.

- **ALL**: This data set consists of the test results for all generated mutants. It is used to assess the defect detection capabilities of unit and integration tests for a large data set with many different defects that are integrated.

- **DISJ**: This data set consists of the test results for the set of disjoint mutants (Section 4.2.6. It is used to gain insights into the defect detection capabilities of unit and integration tests for defects that are "hard to kill" [42].

6.2.2. Analysis Procedure

We executed the following analysis for both data sets presented in Section 6.2.1.

1. **Calculate the Number of Detected Defects:** We sum up the number of all detected defects. We consider a *killed* mutant as a detected defect. The results for test cases that are executed with different parameters are combined and analyzed as one test case. Furthermore, as the algorithm applied to create the disjoint mutant set is non-deterministic (Algorithm 4.1), we repeated the analysis process using the disjoint mutant set 10 times and took the average of all 10 runs for the number of detected defects.

2. **Calculate the Sum of Detected Defects for each Test Level:**
   - **For analyzing the Overall Effectiveness:** We check for each detected defect, if it was detected by an unit or an integration test. We then sum up the number of detected defects for each test set $X \in \{U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}, U_{DEV}, I_{DEV}\}$ and each of the above mentioned data sets. $DetectedDefects(X, p)$ is defined as the number of defects that are detected by tests within a test set X for a project p.

[10] The mutation testing tool that we used for our analysis (Section 4.2.6) is executed for each test case separately. Therefore, all mutants are generated separately for each test case. But, the mutation testing framework pre-selects mutants against which the test case should run by using the coverage data of the test case [67]. Hence, not all mutants are generated for each test case.

- **For analyzing the Defect-Specific Effectiveness** We divide the detected defects by their type (i.e., computation, data, interface, and logic/control) and sum them up for each test set $X \in \{U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}, U_{DEV}, I_{DEV}\}$ separately. $DetectedDefects(X,t,p)$ is defined as the number of defects of type t that are detected by tests within test set X for project p. This way, we can assess if one test level is more effective in detecting a certain type of defect. We excluded the defect type *Other* from our analysis, as it does not represent a real defect type, but more a type of change that can not be classified as one of the other types (Section 4.2.7).

3. **Normalize by the Number of TestLOC:** The resulting sums from the previous step are normalized by the number of Thousand Test Lines of Code (TestKLOC) to create scores. This normalization step is performed to include the effort that was put into the creation of a test into our analysis. Hence, we define the following sets for each test set $X \in \{U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}, U_{DEV}, I_{DEV}\}$.
   - $SCORE(X) := \{\frac{DetectedDefects(X,p)}{\sum_{x \in X} TestKLOC(X,p)} | p \in Projects\}$ is a set that contains all normalized scores for each project.
   - $SCORE_T(X,t) := \{\frac{DetectedDefects(X,t,p)}{\sum_{x \in X} TestKLOC(X,p)} | p \in Projects\}$ is a set that contains all normalized scores for a defect type t for each project.

4. **Check Preconditions for Statistical Testing:** In the next step we separately check for each *SCORE* set if they follow a normal distribution using the Shapiro-Wilk test. Moreover, we check for equal variances using the Brown-Forsythe test, between the *SCORE* sets of unit and integration tests for both definitions and the developer classification.

5. **Statistical Testing:** Based on the results of the previous step, we chose a fitting two-sided significance test to test for differences between the *SCORE* sets of unit and integration tests. Performing multiple statistical tests on the same data set could increase the overall chance of false discoveries (Type 1 Errors) [115]. Therefore, we decided to apply corrections for multiple comparisons. We use the Bonferroni correction (Section 2.3.7) for all of the statistical hypothesis tests that we made. Overall, we use 30 statistical hypothesis tests: six for the analysis of the overall effectiveness, where we check for differences in the scores for the **ALL** and **DISJ** data sets for the IEEE and ISTQB definition, as well as the developer classification, and 24 for the analysis of the defect-specific effectiveness. Within this analysis we reuse the **ALL** and **DISJ** data sets and check for differences in the scores based on the defect type (i.e., computation, data, interface, logic/control), which results in 24 different tests (two data sets * 4 defect types * 3 definitions/classifications). Hence, the adjusted significance level is $\alpha^* = 0.005/30 = 0.0002$. All concrete p-values and test statistics for these tests are presented in Appendix C.2.

6.2.3. Results

Within this section we present the results of our analysis. This includes the results of the analysis of the overall test effectiveness of unit and integration tests (Section 6.2.3.1), as well as the results of the test effectiveness of unit and integration tests separated by defect type (Section 6.2.3.2).

6.2.3.1. Overall Effectiveness of Unit and Integration Tests

Table 6.5 shows the number of defects that are detected by unit tests, integration tests, and both, together with their score (i.e., number of detected defects per TestKLOC) for the **ALL** and **DISJ** data sets, if the tests are classified by the IEEE definition. In addition, the mean and standard deviation is shown for each column.

Table 6.5 shows that there are projects, where integration tests detect more defects than unit tests (e.g., *commons-beanutils*, *commons-collections*), but also projects where unit tests detect more defects than integration tests (e.g., *commons-codec*). This holds true for both data sets, i.e., **ALL** and **DISJ**. There are nine of overall 17 projects for the **ALL** data set, where the integration test scores are higher than the unit test scores. This is also reflected in the mean of the scores (479.43 for unit test scores, 573.57 for integration test scores). However, there is a large standard deviation within both populations (320.81 for unit test scores, 369.63 for integration test scores). The picture is different for the **DISJ** data set. Here, there are only five projects (i.e., *fastjson*, *google-gson*, *HikariCP*, *joda-time*, and *jsoup*) where the integration test scores are higher than the unit test scores. Furthermore, the mean of the scores for the **DISJ** data set are higher for unit tests than for integration tests (5.66 for unit test scores, 2.42 for integration test scores). Nevertheless, the results are also influenced by the large standard deviation that both samples have (7.72 unit test scores, 2.80 integration test scores). There are also mutants that are detected by both test types. For the **ALL** data set 2129.71 mutants are on average detected by both test types and 121.12 mutants for the **DISJ** data set. Interestingly, there are proportionately more mutants that are detected by both test types for the **DISJ** data set, if we take the number of all mutants in the data set into account.

We performed several tests for the set of scores. For the **ALL** data set the unit test scores, as well as the integration test scores follow a normal distribution. In addition, both score sets have equal variances. The t-test (Section 2.3.5.3) showed, that the t-statistic is not significant at the .0002 critical alpha level. Therefore, we conclude that while there is a difference in the mean of the scores, this difference is not statistically significant.

For the **DISJ** data set we found that only the integration test scores follow a normal distribution. In addition, both samples have equal variances. However, the U-statistic was not significant at the .0002 critical alpha level. Therefore, we conclude that there is no statistically significant difference in the defect detection capabilities between unit and integration tests for the **DISJ** data set using the IEEE definition.

	Killed Mutants						Scores			
	ALL			DISJ			ALL		DISJ	
Project	UT	IT	B	UT	IT	B	UT	IT	UT	IT
commons-beanutils	1292	3682	1001	10	22	75	117.82	82.72	0.91	0.49
commons-codec	5219	1001	1187	72	7	47	847.24	881.16	11.69	6.16
commons-collections	2119	6009	1696	35	41	91	33.61	23.03	0.56	0.16
commons-io	3308	2683	1554	63	15	144	190.72	203.49	3.63	1.14
commons-lang	20765	5426	4537	434	12	216	565.53	500.51	11.82	1.11
commons-math	3808	72914	9353	27	261	511	352.20	461.77	2.50	1.65
druid	618	72285	486	18	388	46	1056.41	1011.60	30.77	5.43
fastjson	836	29031	3009	4	443	376	424.80	702.73	2.03	10.72
google-gson	809	3816	1808	4	22	27	370.25	343.32	1.83	1.98
guice	164	7544	360	1	50	28	618.87	557.74	3.77	3.70
HikariCP	106	2038	120	0	3	11	364.26	348.50	0.00	0.51
jackson-core	529	19198	951	6	24	57	1079.59	1092.84	12.24	1.37
jfreechart	1228	32912	996	22	69	78	464.80	1053.52	8.33	2.21
joda-time	505	24749	1056	6	295	123	179.72	295.55	2.14	3.52
jsoup	128	8754	1030	0	3	39	181.56	743.63	0.00	0.25
mybatis-3	626	7473	4631	4	7	149	432.02	261.50	2.76	0.24
zxing	1362	16393	2430	2	7	41	870.84	1187.12	1.28	0.51
Mean	2554.24	18582.82	2129.71	41.65	98.18	121.12	479.43	573.57	5.66	2.42
StDev	4906.60	22558.71	2274.67	103.38	147.71	134.68	320.81	369.63	7.72	2.80

Table 6.5.: Number of mutants that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) together with their scores for the **ALL** and **DISJ** data sets. The tests are classified into unit and integration test according to the **IEEE** definition.

	Killed Mutants						Scores			
	ALL			DISJ			ALL		DISJ	
Project	UT	IT	B	UT	IT	B	UT	IT	UT	IT
commons-beanutils	110	5569	296	2	82	23	117.27	102.11	2.13	1.50
commons-codec	1365	5368	674	35	33	58	539.95	1125.84	13.84	6.92
commons-collections	1296	7536	992	25	69	73	82.20	24.45	1.59	0.22
commons-io	1030	5810	705	25	108	89	309.87	213.56	7.52	3.97
commons-lang	13767	12663	4298	324	91	247	712.87	448.30	16.78	3.22
commons-math	763	80916	4396	21	396	382	212.30	490.04	5.84	2.40
druid	379	72718	292	17	395	40	832.97	1015.81	37.36	5.52
fastjson	286	31600	990	4	681	138	460.55	740.76	6.44	15.96
google-gson	228	4516	1689	1	31	21	175.65	376.27	0.77	2.58
guice	10	8028	30	0	70	9	175.44	584.53	0.00	5.10
HikariCP	46	2141	77	0	7	7	312.93	357.31	0.00	1.17
jackson-core	370	19492	816	3	38	46	1128.05	1099.44	9.15	2.14
jfreechart	861	33325	950	20	83	66	436.17	1044.41	10.13	2.60
joda-time	69	26019	222	6	367	51	144.05	302.30	12.53	4.26
jsoup	68	9067	777	0	16	26	140.21	756.09	0.00	1.33
mybatis-3	99	11979	652	3	80	77	191.12	405.96	5.79	2.71
zxing	254	18738	1193	1	12	37	356.24	1278.17	1.40	0.82
Mean	1235.35	20910.88	1120.53	28.65	150.53	81.76	372.23	609.73	7.72	3.67
StDev	3259.85	23109.55	1286.23	76.94	190.34	96.40	289.18	387.13	9.27	3.63

Table 6.6.: Number of mutants that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) together with their scores for the **ALL** and **DISJ** data sets. The tests are classified into unit and integration test according to the **ISTQB** definition.

Table 6.6 shows the number of defects that are detected by unit tests, integration tests, and both, together with their score (i.e., number of detected defects per TestKLOC) for the **ALL** and **DISJ** data sets, if the tests are classified by using the ISTQB definition. In addition, the mean and standard deviation is shown for each column. Table 6.6 shows that there is only one project where the unit tests detect more defects than the integration tests for the **ALL** data set (i.e., *commons-lang*). For all the other projects and the **ALL** data set the integration tests detect more defects than the unit tests. Nevertheless, there are some projects where the unit test score is higher than the integration test score (e.g., *commons-beanutils*, *commons-collections*, *commons-io*). This is also highlighted by the mean of the scores for the **ALL** data set (372.23 for unit test scores, 609.73 for integration test scores). However, these results are influenced by the high standard deviation (289.18 for unit test scores, 387.13 for integration test scores). This is different for the **DISJ** data set, as shown in Table 6.6. There are now two projects, where unit tests detect more defects than integration tests (i.e., *commons-codec* and *commons-lang*). However, a comparison of the scores shows that the unit test scores are higher than the integration test scores for 12 projects. Furthermore, for the **DISJ** data set the mean of the scores is higher for unit tests than for integration tests (7.72 for unit tests, 3.67 for integration tests). Nevertheless, this is also influenced by the high standard deviation (9.27 for unit tests, 3.63 for integration tests). On average, 1120.53 mutants are detected by both test types for the **ALL** and 81.76 mutants for the **DISJ** data sets. These numbers are lower than the numbers presented in Table 6.5.

We performed several tests for the sets of scores. For the **ALL** data set we found that the unit test scores, as well as the integration test scores follow a normal distribution. Furthermore, they both have equal variances. However, the t-statistic is not significant at the .0002 critical alpha level. Hence, we conclude that while there is a difference in the mean of the scores, this difference is not statistically significant. These results are similar to the tests that we performed on the scores for the **DISJ** data set. With regard to the scores, the unit tests and the integration tests scores follow a normal distribution and both samples have equal variances. The t-statistic is not significant at the .0002 critical alpha level. Hence, we conclude that while there is a difference in the mean of the scores, this difference is not statistically significant.

Table 6.7 shows the number of defects that are detected by unit tests, integration tests, and both, together with their score (i.e., number of detected defects per TestKLOC) for the **ALL** and **DISJ** data sets, if the tests are classified according to the developer classification. In addition, the mean and standard deviation is shown for each column. Table 6.7 shows that there are five of overall 17 projects where unit tests detect more defects than the integration tests for the **ALL** data set (i.e., *commons-codec*, *commons-collections*, *commons-lang*, *commons-math*, *jfreechart*). For all the other projects and the **ALL** data set the integration tests detect more defects than the unit tests. Furthermore, on average 5881.76 defects are detected by both test types for the **ALL** data set. There are nine projects, where the unit test scores for the **ALL** data set depicted in Table 6.7 are higher than the integration test scores. That unit and integration tests have similar scores for the **ALL** data set over the projects

	Killed Mutants						Scores			
	ALL			DISJ			ALL		DISJ	
Project	UT	IT	B	UT	IT	B	UT	IT	UT	IT
commons-beanutils	0	5975	0	0	107	0	0.00	106.93	0.00	1.91
commons-codec	4769	740	1898	100	2	24	710.10	1213.11	14.89	3.28
commons-collections	8735	200	889	112	3	52	35.96	2.34	0.46	0.04
commons-io	2258	4425	867	66	81	75	275.50	197.62	8.05	3.62
commons-lang	22841	4048	4102	500	33	129	588.43	420.13	12.88	3.43
commons-math	56762	6339	22974	187	6	606	396.85	246.90	1.31	0.23
druid	2876	53609	16904	17	35	400	465.15	813.97	2.75	0.53
fastjson	869	23277	8730	2	75	746	308.59	573.10	0.71	1.85
google-gson	1259	2459	2715	3	4	46	453.69	237.52	1.08	0.39
guice	503	2853	4712	1	7	71	144.42	276.78	0.29	0.68
HikariCP	292	857	1115	1	3	10	657.66	150.06	2.25	0.53
jackson-core	1661	15032	3985	5	21	61	892.53	928.13	2.69	1.30
jfreechart	30907	480	3749	141	1	27	944.62	411.66	4.31	0.86
joda-time	5905	9841	10564	32	2	390	220.77	164.27	1.20	0.03
jsoup	2232	2403	5277	3	1	38	273.66	553.18	0.37	0.23
mybatis-3	1421	2246	9063	5	0	155	143.97	111.40	0.51	0.00
zxing	644	17095	2446	0	35	15	809.05	1172.74	0.00	2.40
Mean	8466.71	8934.06	5881.76	69.12	24.47	167.35	430.64	445.87	3.16	1.25
StDev	15083.86	13257.78	6184.33	125.48	33.13	226.75	294.86	377.33	4.52	1.26

Table 6.7.: Number of mutants that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) together with their scores for the **ALL** and **DISJ** data sets. The tests are classified into unit and integration test according to the developer classification.

can be seen on the mean, which is close together (i.e., 430.64 for the unit test scores and 445.87 for the integration test scores). However, these results are influenced by the high standard deviation (294.86 for unit test scores, 377.33 for integration test scores). The picture is different for the **DISJ** data set. The number of projects, where unit tests detect more defects than integration tests is now similar to the number of projects where it is vice versa (i.e, 8:9). In addition, there are on average 167.35 defects found by both test types for the **DISJ** data set, which is larger compared to the **ALL** data set. However, a comparison of the scores shows that the unit test scores are higher than the integration test scores for 13 projects. Furthermore, the difference of the mean scores (i.e., 3.16 for unit tests, 1.26 for integration tests) between unit and integration tests is now larger for the **DISJ** data set. Nevertheless, this is also influenced by the standard deviation (4.52 for unit tests, 1.26 for integration tests).

We performed several tests for the sets of scores. For the **ALL** data set we found that the unit test scores, as well as the integration test scores follow a normal distribution. Furthermore, they both have equal variances. However, the t-statistic is not significant at the .0002 critical alpha level. Hence, we conclude that there is no statistically significant difference in the mean of the scores. These results are similar to the tests that we performed on the scores for the **DISJ** data set. With regard to the scores, only the integration tests scores follow a normal distribution, but both samples have equal variances. The U-statistic is not significant at the .0002 critical alpha level. Hence, our result is the same as with the **ALL** data set, i.e., there is no statistically significant difference in the defect detection capabilities between unit and integration tests, if they are classified according to the developer classification.

Answer to RQ 2.2 (Overall Effectiveness): Our results highlighted in tables 6.5, 6.6, and 6.7 and the statistical tests that we performed show that there is no statistically significant difference in the overall effectiveness between unit and integration tests. Neither for any of the defect data sets (i.e., **ALL** and **DISJ**), nor for any of the tested definitions (i.e., IEEE and ISTQB) or the developer classification. Hence, neither the test level definition used nor the set of mutants have an influence on the results. However, our data shows that unit tests are (in the mean) more effective in detecting the "hard to kill" mutants than integration tests, but not significantly. This holds true for both test level definitions. This is in line with Papadakis et al. [42] who stated that mutation-based assessment metrics can change if disjoint mutants are considered. Furthermore, our data indicates that there are differences between projects (highlighted by the large standard deviation), as some projects have unit tests that are more effective (e.g., *commons-lang*), while in other projects the integration tests are more effective (e.g., *commons-math*). Moreover, we also see that there are mutants that are detected by both test types, while the numbers differ between the analyzed test sets. Hence, it seems that both types test similar parts in the software.

	COMP.		DATA		INT.		L/C	
Project	UT	IT	UT	IT	UT	IT	UT	IT
commons-beanutils	3.83	2.29	6.29	3.30	38.39	32.44	69.31	44.69
commons-codec	71.27	22.01	134.42	66.02	252.44	344.19	389.12	448.94
commons-collections	2.24	1.25	2.98	1.11	8.31	7.05	20.08	13.62
commons-io	11.01	15.09	20.12	19.34	57.94	64.77	101.64	104.29
commons-lang	24.59	17.06	61.36	44.46	125.25	162.62	354.29	276.36
commons-math	28.39	38.68	41.90	76.47	86.57	129.81	195.15	215.08
druid	133.33	57.56	169.23	55.92	194.87	396.17	558.97	501.95
fastjson	31.50	50.47	66.57	109.65	63.01	213.72	263.72	328.86
google-gson	66.36	17.09	59.04	14.22	65.45	115.61	174.37	157.17
guice	11.32	37.93	18.87	31.05	226.42	223.94	362.26	264.82
HikariCP	37.80	30.61	65.29	28.73	151.20	125.68	109.97	163.13
jackson-core	87.76	153.07	136.73	154.32	293.88	195.42	561.22	589.97
jfreechart	36.34	76.47	71.16	92.93	75.32	290.01	281.98	593.41
joda-time	7.47	13.76	19.93	21.91	65.12	100.98	87.19	158.89
jsoup	5.67	32.45	19.86	58.78	41.13	280.33	114.89	372.07
mybatis-3	15.87	13.89	18.63	12.25	193.24	119.15	204.28	116.21
zxing	41.56	81.83	190.54	211.46	203.32	282.42	435.42	611.12
Mean	36.25	38.91	64.88	58.94	125.99	181.43	251.99	291.80
StDev	35.45	37.67	58.56	56.96	86.22	110.24	168.34	197.19

Table 6.8.: Scores for unit and integration tests, classified by the **IEEE** definition, for the **ALL** data set and separated by defect type.

6.2.3.2. Defect-Specific Effectiveness of Unit and Integration Tests

Tables 6.8 and 6.9 show the scores (i.e., number of detected defects per TestKLOC) of unit and integration tests, separated by the type of defect that they have detected for the IEEE definition, for the **ALL** and **DISJ** data sets respectively. In addition, the mean and standard deviation is shown for each column of the tables. Additional tables and visualizations, including the number of detected defects separated by test level and defect type, can be found in Appendix D.

Table 6.8 depicts the scores for unit and integration tests, as classified by the IEEE definition, for the data set **ALL**. This table highlights that the mean scores of integration tests are higher than the mean scores of unit tests for each defect type, except DATA defects. However, they also have a higher standard deviation. This data shows that integration tests are more effective (i.e., the scores are higher) for any defect type, except DATA defects. Some of the differences between the projects are made more evident in Table 6.8. For some projects, the results are as expected (i.e., integration tests are more effective in detecting interface defects, while unit tests are more effective in detecting other defects), while for other projects it is vice versa (e.g., *jackson-core*).

Table 6.9 shows the scores for unit and integration tests, as classified by the IEEE definition, for the data set **DISJ**. This table shows a different picture than Table 6.8. Integration tests are less effective in detecting any defect type than unit tests, i.e., the mean of the

	COMP.		DATA		INT.		L/C	
Project	UT	IT	UT	IT	UT	IT	UT	IT
commons-beanutils	0.09	0.00	0.09	0.02	0.18	0.09	0.55	0.38
commons-codec	0.97	1.76	0.97	0.00	2.60	2.64	7.14	1.76
commons-collections	0.00	0.01	0.06	0.00	0.14	0.03	0.35	0.11
commons-io	0.23	0.15	0.40	0.15	1.33	0.15	1.67	0.68
commons-lang	0.54	0.09	1.03	0.28	2.42	0.46	7.82	0.28
commons-math	0.18	0.30	0.09	0.16	0.37	0.17	1.85	1.02
druid	3.42	0.66	3.42	0.32	6.84	1.64	17.09	2.81
fastjson	0.00	1.02	0.00	1.21	0.00	2.54	2.03	5.95
google-gson	0.92	0.00	0.00	0.27	0.00	0.18	0.92	0.81
guice	0.00	0.37	0.00	0.44	0.00	1.11	3.77	1.77
HikariCP	0.00	0.00	0.00	0.17	0.00	0.17	0.00	0.17
jackson-core	0.00	0.34	0.00	0.06	2.04	0.23	10.20	0.74
jfreechart	1.51	0.35	0.76	0.00	1.14	0.54	4.92	1.31
joda-time	0.00	0.11	0.00	0.17	0.71	0.60	1.42	2.65
jsoup	0.00	0.00	0.00	0.08	0.00	0.08	0.00	0.08
mybatis-3	0.00	0.14	0.00	0.00	2.07	0.03	0.69	0.07
zxing	0.64	0.07	0.00	0.14	0.00	0.00	0.64	0.29
Mean	0.50	0.32	0.40	0.20	1.17	0.63	3.59	1.23
StDev	0.88	0.46	0.86	0.29	1.74	0.85	4.62	1.49

Table 6.9.: Scores for unit and integration tests, classified by the **IEEE** definition, for the **DISJ** data set and separated by defect type.

integration test scores are lower than the mean of the unit test scores. However, the standard deviation of the integration test scores is lower than the standard deviation of the unit test scores. Nevertheless, Table 6.9 also depicts that some defects are only detected by integration tests (e.g., interface defects for the projects *fastjson*, *google-gson*, *guice*, *HikariCP*, *jsoup*) or only unit tests (e.g., computation defects for the projects *commons-beanutils*, *google-gson*).

We performed significance tests between the scores of unit and integration tests for each defect type and for the data sets **ALL** and **DISJ**. None of the test statistics was significant for the .0002 critical alpha level. Therefore, we can conclude that neither unit nor integration tests are more effective in detecting any kind of defect type, if we classify our tests according to the IEEE definition.

Tables 6.10 and 6.11 show the scores (i.e., number of detected defects per TestKLOC) of unit and integration tests, separated by the type of defect that they have detected for the ISTQB definition, for the **ALL** and **DISJ** data sets, respectively. In addition, the mean and standard deviation is shown for each column of the tables. Additional tables, including the number of detected defects separated by test level and defect type can be found in Appendix D.

Table 6.10 depicts the scores for unit and integration tests, as classified by the ISTQB definition, for the data set **ALL**. This table highlights that the mean scores of integration

	COMP.		DATA		INT.		L/C	
Project	UT	IT	UT	IT	UT	IT	UT	IT
commons-beanutils	6.40	3.21	11.73	4.14	29.85	38.14	69.30	56.62
commons-codec	28.88	105.70	81.49	176.17	158.23	301.80	271.36	542.16
commons-collections	5.58	1.39	7.80	1.25	18.27	7.40	50.55	14.41
commons-io	38.51	11.17	43.92	19.26	81.83	67.71	145.61	115.38
commons-lang	32.16	17.59	79.33	42.31	140.17	134.92	461.16	253.48
commons-math	12.24	41.24	20.31	79.90	45.91	135.89	133.83	231.35
druid	112.09	58.03	147.25	56.56	129.67	397.07	443.96	504.15
fastjson	32.21	54.15	61.19	117.11	35.43	217.56	331.72	351.91
google-gson	13.10	27.25	30.05	21.83	43.14	115.31	89.37	172.30
guice	0.00	39.10	0.00	33.49	70.18	231.83	105.26	280.11
HikariCP	34.01	31.38	54.42	30.21	136.05	129.67	88.44	165.72
jackson-core	70.12	153.42	158.54	154.77	314.02	197.08	585.37	594.11
jfreechart	28.88	76.13	67.38	92.89	67.88	286.20	272.04	588.50
joda-time	2.09	14.02	14.61	22.81	31.32	102.18	96.03	163.27
jsoup	6.19	33.61	18.56	60.12	37.11	284.61	78.35	377.75
mybatis-3	15.44	24.10	15.44	18.00	88.80	177.41	71.43	186.46
zxing	22.44	84.52	71.53	227.90	70.13	303.14	192.15	662.35
Mean	27.08	45.65	51.97	68.16	88.12	184.00	205.05	309.41
StDev	28.03	40.18	46.39	65.68	72.66	106.87	163.18	202.59

Table 6.10.: Scores for unit and integration tests, classified by the **ISTQB** definition, for the **ALL** data set and separated by defect type.

tests are higher than the mean scores of unit tests for each defect type. However, they also have a higher standard deviation. This data shows that integration tests are more effective (i.e., the scores are higher) for any defect type. Moreover, for some projects the results are as expected (i.e., integration tests are more effective in detecting interface defects, while unit tests are more effective in detecting other defects), while for other projects it is vice versa (e.g., *jackson-core*). Overall, the results for tables 6.8 and 6.10 are similar, highlighting that the used definition (i.e., IEEE or ISTQB) does not has an influence on the results.

Table 6.11 shows the scores for unit and integration tests, as classified by the ISTQB definition, for the data set **DISJ**. This table shows a different picture than Table 6.10, but a similar than Table 6.9. Integration tests are less effective in detecting any defect type than unit tests, i.e., the mean of the integration test scores are lower than the mean of the unit test scores. However, the standard deviation of the integration test scores is lower than the standard deviation of the unit test scores. The similarity between tables 6.9 and 6.11 indicates that the test level definition being used (i.e., IEEE or ISTQB) has no influence on our data. Nevertheless, Table 6.11 highlights that some defects are only detected by integration tests (e.g., interface defects for the projects *commons-beanutils*, *fastjson*, *google-gson*, *guice*, *HikariCP*, *jsoup*, *zxing*) or only unit tests (e.g., data defects for the projects *commons-codec*, *commons-collections*).

We performed significance tests between the scores of unit and integration tests for each

	COMP.		DATA		INT.		L/C	
Project	UT	IT	UT	IT	UT	IT	UT	IT
commons-beanutils	1.07	0.06	0.00	0.06	0.00	0.29	1.07	1.10
commons-codec	0.79	1.89	1.98	0.00	1.98	1.68	9.10	3.36
commons-collections	0.00	0.01	0.25	0.00	0.25	0.05	1.08	0.16
commons-io	0.30	0.33	1.50	0.18	1.50	1.47	4.21	1.98
commons-lang	0.78	0.25	1.55	0.32	2.64	1.31	11.81	1.35
commons-math	0.28	0.38	0.00	0.22	0.83	0.28	4.73	1.51
druid	4.40	0.66	4.40	0.35	8.79	1.66	19.78	2.85
fastjson	0.00	1.81	0.00	1.78	0.00	3.14	6.44	9.24
google-gson	0.00	0.17	0.00	0.42	0.00	0.25	0.77	0.92
guice	0.00	0.51	0.00	0.44	0.00	1.67	0.00	2.48
HikariCP	0.00	0.00	0.00	0.17	0.00	0.67	0.00	0.33
jackson-core	0.00	0.39	0.00	0.06	3.05	0.23	6.10	1.47
jfreechart	2.03	0.34	1.01	0.03	1.01	0.72	6.08	1.50
joda-time	0.00	0.13	0.00	0.20	4.18	0.70	8.35	3.24
jsoup	0.00	0.17	0.00	0.25	0.00	0.42	0.00	0.50
mybatis-3	0.00	0.17	0.00	0.07	3.86	1.15	1.93	1.32
zxing	1.40	0.20	0.00	0.14	0.00	0.07	0.00	0.41
Mean	0.65	0.44	0.63	0.27	1.65	0.93	4.79	1.98
StDev	1.14	0.56	1.18	0.41	2.34	0.82	5.33	2.11

Table 6.11.: Scores for unit and integration tests, classified by the **ISTQB** definition, for the **DISJ** data set and separated by defect type.

defect type and for the data sets **ALL** and **DISJ**. None of the test statistics was significant for the .0002 critical alpha level. Therefore, we can conclude that neither unit nor integration tests are more effective in detecting any kind of defect type, if we classify our tests according to the ISTQB definition. This is in line with our results for the IEEE definition.

Tables 6.12 and 6.13 show the scores (i.e., number of detected defects per TestKLOC) of unit and integration tests, separated by the type of defect that they have detected for developer classification and the **ALL** and **DISJ** data sets, respectively. In addition, the mean and standard deviation is shown for each column of the tables. Additional tables, including the number of detected defects separated by test level and defect type can be found in Appendix D.

Table 6.12 depicts the scores for unit and integration tests, as classified by the developers of the projects, for the data set **ALL**. This table shows a different picture than tables 6.8 and 6.10. Here, the mean scores of unit tests is higher for computation defects, while the mean scores of integration tests is higher for data and interface defects. The difference between the mean scores of unit and integration tests for logic/control defects is only .69. However, these scores also have the highest standard deviation. This data shows a mixed picture, i.e., it seems that unit tests are better for certain defect types than integration tests and vice versa, with the exception of logic/control defects. This mixed picture can also be seen, if we have a closer look at the differences between projects for each defect type. For

	COMP.		DATA		INT.		L/C	
Project	UT	IT	UT	IT	UT	IT	UT	IT
commons-beanutils	0.00	3.31	0.00	4.78	0.00	38.89	0.00	59.95
commons-codec	56.28	57.38	102.14	211.48	215.90	400.00	335.77	544.26
commons-collections	2.19	0.12	2.23	0.01	10.16	0.83	21.37	1.38
commons-io	28.79	9.24	34.65	17.37	83.70	61.94	128.36	109.01
commons-lang	27.59	8.82	58.58	44.11	136.90	126.21	365.36	240.89
commons-math	32.08	15.70	65.23	48.06	113.57	76.61	184.08	106.49
druid	45.93	43.39	39.46	43.50	158.50	331.05	221.25	396.03
fastjson	11.72	45.03	20.60	94.89	85.23	163.01	191.05	270.12
google-gson	51.89	12.46	57.30	11.88	115.68	81.04	224.86	109.63
guice	7.18	13.00	9.76	16.30	47.37	112.44	80.10	135.04
HikariCP	60.81	9.28	92.34	9.81	261.26	61.29	240.99	69.69
jackson-core	113.38	131.21	112.31	136.64	192.91	162.94	473.94	497.28
jfreechart	69.47	12.01	91.32	28.30	235.58	187.82	547.57	183.53
joda-time	8.26	4.39	18.88	10.70	66.29	67.07	127.34	82.09
jsoup	7.48	32.00	21.46	46.96	119.05	200.51	125.67	273.71
mybatis-3	5.17	6.10	5.88	6.15	64.84	49.65	68.09	49.50
zxing	28.89	81.77	157.04	211.57	209.80	270.36	413.32	608.77
Mean	32.77	28.54	52.30	55.44	124.51	140.69	220.54	219.85
StDev	30.51	34.74	45.53	68.48	78.09	109.51	158.64	187.09

Table 6.12.: Scores for unit and integration tests, classified according to the developers, for the **ALL** data set and separated by defect type.

example, there are 9 projects where the unit test scores are higher than the integration tests scores for interface defects and 8 projects where it is vice versa. Overall, the results for tables 6.8, 6.10, and 6.12 are different, highlighting that there is a difference between the actual definitions of the IEEE and ISTQB and current development practices.

Table 6.13 shows the scores for unit and integration tests, as classified by the developers of the projects, for the data set **DISJ**. This table shows a different picture than Table 6.12, but a similar than tables 6.9 and 6.11. Integration tests are less effective in detecting any defect type than unit tests, i.e., the mean of the integration test scores are lower than the mean of the unit test scores. However, the standard deviation of the integration test scores is lower than the standard deviation of the unit test scores. The similarity between tables 6.9, 6.11, and 6.13 indicates that the test level definition being used (i.e., IEEE, ISTQB, or reusing the developer classification) has no influence on our data. Nevertheless, Table 6.13 highlights that some defects are only detected by integration tests (e.g., interface defects for the projects *commons-beanutils*, *fastjson*, *google-gson*, *guice*, *zxing*) or only unit tests (e.g., data defects for the projects *commons-codec*, *commons-collections*, *commons-math*, *joda-time*).

We performed significance tests between the scores of unit and integration tests for each defect type and for the data sets **ALL** and **DISJ**. None of the test statistics was significant for the .0002 critical alpha level. Therefore, we can conclude that neither unit nor integration

	COMP.		DATA		INT.		L/C	
Project	UT	IT	UT	IT	UT	IT	UT	IT
commons-beanutils	0.00	0.09	0.00	0.05	0.00	0.32	0.00	1.45
commons-codec	1.34	0.00	0.74	0.00	2.38	3.28	10.42	0.00
commons-collections	0.02	0.00	0.02	0.00	0.10	0.00	0.32	0.04
commons-io	0.73	0.40	0.85	0.18	3.54	0.89	2.93	2.14
commons-lang	0.90	0.00	1.19	0.21	2.55	1.35	8.27	1.87
commons-math	0.15	0.00	0.12	0.00	0.18	0.08	0.86	0.16
druid	0.32	0.05	0.32	0.03	0.65	0.18	1.46	0.27
fastjson	0.00	0.12	0.00	0.22	0.00	0.37	0.71	1.13
google-gson	0.36	0.00	0.00	0.10	0.00	0.10	0.72	0.19
guice	0.00	0.10	0.00	0.10	0.00	0.19	0.29	0.29
HikariCP	0.00	0.00	0.00	0.18	2.25	0.18	0.00	0.18
jackson-core	0.54	0.19	0.00	0.00	0.54	0.12	1.61	0.99
jfreechart	0.76	0.00	0.28	0.86	0.70	0.00	2.57	0.00
joda-time	0.04	0.00	0.04	0.00	0.19	0.02	0.93	0.02
jsoup	0.00	0.00	0.00	0.00	0.25	0.23	0.12	0.00
mybatis-3	0.00	0.00	0.00	0.00	0.30	0.00	0.20	0.00
zxing	0.00	0.34	0.00	0.34	0.00	0.21	0.00	1.51
Mean	0.30	0.08	0.21	0.13	0.80	0.44	1.85	0.60
StDev	0.41	0.13	0.37	0.21	1.13	0.81	2.98	0.74

Table 6.13.: Scores for unit and integration tests, classified according to the developers, for the **DISJ** data set and separated by defect type.

tests are more effective in detecting any kind of defect type, if we reuse the developer classification. This is in line with our results for the IEEE and ISTQB definition.

Answer to RQ 2.2 (Defect-Specific Effectiveness): Tables 6.8, 6.9, 6.10, 6.11, 6.12, and 6.13 indicate that there are differences in the effectiveness between unit and integration tests for different defect types, if we compare the mean scores of unit and integration tests. However, the results are different from project to project. Furthermore, we found that the differences are minimal, if we reuse the developer classification for our separation into unit and integration tests. For some projects, our results are as expected, as integration tests are more effective in detecting interface defects, but for some projects it is vice versa (i.e., integration tests are more effective in detecting any kind of defect except interface defects). Furthermore, our results are not consistent over the **ALL** and **DISJ** data sets. This is an indication that unit tests are more effective in detecting "hard to kill" defects, which is in line with the result of our overall effectiveness analysis. However, while the mean scores are different between unit and integration tests, our statistical tests highlight that these differences are not statistically significant. Interestingly, our results are consistent for the different test level definitions that we used, i.e., the definitions of the IEEE and ISTQB, as well as the developer classification, indicating that the definition being used has no influence on our results.

6.3. Evaluation of RQ 2.3: Test Defect-Locality

One difference mentioned in the literature is that the source of the defect can be easier identified if a unit test failed instead of, e.g., an integration test. We evaluate this difference within this RQ by using our defect-locality metric (Section 4.2.8) that is designed as a proxy metric to evaluate this difference. Therefore, we present our data set description (Section 6.3.1), our analysis procedure (Section 6.3.2), as well as our results (Section 6.3.3) for this RQ within this section.

6.3.1. Data set Description

For the analysis of this RQ, we reuse the data set described in Section 6.2.1, as our approach to extract the defect-locality needs the collected mutation data (Section 4.2.8).

6.3.2. Analysis Procedure

For all of the mutant data sets presented in Section 6.2.1, we executed the following analysis process.

1. **Calculate the Average Defect-Locality:** As a first step, we calculate the average defect-locality for each test set $X \in \{U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}, U_{DEV}, I_{DEV}\}$ by summing up the $dl(x,d)$ values for each $x \in X$ and $d \in \{ALL, DISJ\}$ and dividing it by the number of dl values that we aggregate. These values are summarized in different sets that are defined as $DL_{AVG}(X) := \{dl_{AVG}(X,p) | p \in Projects\}$ for each test set $X \in \{U_{IEEE}, I_{IEEE}, U_{ISTQB}, I_{ISTQB}, U_{DEV}, I_{DEV}\}$, where $dl_{AVG}(X,p)$ is the average of all defect-locality values for all $x \in X$ and all mutants (i.e., the **ALL** or **DISJ** set) for a project p.

2. **Check Preconditions for Statistical Testing:** We check for each DL_{AVG} set if it follows a normal distribution using the Shapiro-Wilk test. Moreover, we check for equal variances between $DL_{AVG}(U_{IEEE})$ and $DL_{AVG}(I_{IEEE})$, $DL_{AVG}(U_{ISTQB})$ and $DL_{AVG}(I_{ISTQB})$, as well as $DL_{AVG}(U_{DEV})$ and $DL_{AVG}(I_{DEV})$ using the Brown-Forsythe test.

3. **Statistical Testing:** Based on the results of the previous step, we chose a fitting one-sided significance test to test for differences between $DL_{AVG}(U_{IEEE})$ and $DL_{AVG}(I_{IEEE})$, $DL_{AVG}(U_{ISTQB})$ and $DL_{AVG}(I_{ISTQB})$, as well as $DL_{AVG}(U_{DEV})$ and $DL_{AVG}(I_{DEV})$ to evaluate if the average defect-locality of unit tests is smaller than the average defect-locality of integration tests.

4. **Effect Size:** If our statistical test showed a difference in the defect-locality between unit and integration tests, we assess the effect size by calculating Cohen's d (Sec-

tion 2.3.6) between them to measure the practical relevance of the observed significant result.

6.3.3. Results

Table 6.14 summarizes the dl_{AVG} values for each analyzed test set and the **ALL** and **DISJ** mutant data sets. For the **ALL** data set, the results show that, except for one project (i.e., *commons-lang*), the dl_{AVG} values for unit tests are always lower than for integration tests. This holds true for both definitions. For the developer classification, there are two projects where it is vice versa, i.e., *commons-codec* and *commons-io*. The mean is lower for unit tests (i.e., 2.99 (IEEE), 1.88 (ISTQB), and 5.94 (DEV)) than for integration tests (i.e., 7.63 (IEEE), 7.67 (ISTQB), and 11.01 (DEV)) for both definitions and the developer classification. However, the standard deviation is rather high for all integration test sets (i.e., 4.10 (IEEE), 3.99 (ISTQB), and 7.62 (DEV)) compared to the unit test sets (1.96 (IEEE), 0.58 (ISTQB), and 7.62 (DEV)). This gives a first hint, that the defect-locality is (on average) smaller for unit tests resulting in a clearer pinpointing of defects, at least for the **ALL** data set.

The results for the **DISJ** mutant data set, depicted in Table 6.14, are similar to the **ALL** data set described above. The unit tests of the *commons-lang* project have a higher average defect-locality than its integration tests. However, this is the only project where this is the case for the IEEE and ISTQB definitions. For the developer classification, the projects *commons-codec* and *commons-collections* have unit tests, where the dl_{AVG} values are higher for their unit than for their integration tests. The mean shows, that the defect-locality of unit tests is on average lower (i.e., 2.78 (IEEE), 1.85 (ISTQB), and 5.25 (DEV)) than for integration tests (i.e., 7.15 (IEEE), 7.18 (ISTQB), and 10.24 (DEV)) for both definitions and the developer classification, which is a similar result as above. Furthermore, the standard deviation of the dl_{AVG} values for the unit test sets is lower (i.e., 1.61 (IEEE), 0.50 (ISTQB), and 3.08 (DEV)) than for the integration test sets (i.e., 3.47 (IEEE), 3.37 (ISTQB), and 6.68 (DEV)).

Figure 6.2 shows box-plots of the dl_{AVG} values. The left part of Figure 6.2 shows the dl_{AVG} values for the **ALL** mutant data set and the right part the values for the **DISJ** data sets. This figure indicates, that the general trend is similar. The box-plots highlight that the median is always lower for unit tests than for integration tests, regardless of the test set or mutant data set. Furthermore, the 25%-quantile of the integration test box-plot is always higher than the 75%-quantile of the unit test box-plot. This indicates that unit tests have indeed a smaller defect-locality than integration tests.

While we do see the same difference in the data for both mutant data sets (i.e., **ALL** and **DISJ**) in Table 6.14 and visually in Figure 6.2, we need to assess it statistically using statistical hypothesis tests. The concrete p-values and test statistics for each statistical test are reported in Appendix C.2. The Shapiro-Wilk tests showed, that all sets for both mutant data sets except the $DL_{AVG}(U_{IEEE})$ set follow a normal distribution. However, only the

	ALL						DISJ					
Project	U_{IEEE}	I_{IEEE}	U_{ISTQB}	I_{ISTQB}	U_{DEV}	I_{DEV}	U_{IEEE}	I_{IEEE}	U_{ISTQB}	I_{ISTQB}	U_{DEV}	I_{DEV}
commons-beanutils	3.47	6.06	1.66	5.87	0.00	5.85	6.63	8.07	1.50	7.98	0.00	7.97
commons-codec	3.49	3.76	2.09	3.88	3.55	3.38	3.10	3.64	1.86	3.78	3.22	3.20
commons-collections	2.08	3.11	1.81	3.03	2.91	3.04	3.05	4.90	2.50	4.80	4.57	3.97
commons-io	2.60	3.42	2.41	3.17	3.12	3.11	2.41	2.97	1.92	2.80	2.28	2.85
commons-lang	6.38	4.45	2.08	6.48	4.02	7.11	6.03	4.94	1.97	6.58	3.62	7.11
commons-math	3.49	12.91	2.45	12.70	14.32	21.11	3.11	6.28	2.73	6.24	6.26	12.67
druid	1.71	9.59	1.04	9.59	6.92	18.72	1.36	9.54	1.24	9.54	8.03	17.24
fastjson	1.68	6.92	1.23	6.88	5.28	11.59	1.64	6.30	1.42	6.24	4.77	11.07
google-gson	3.07	6.44	2.28	6.40	3.87	6.47	1.28	5.64	1.16	5.58	2.86	5.70
guice	2.01	12.08	1.82	12.07	11.43	23.07	1.76	12.73	1.90	12.71	11.30	25.45
HikariCP	1.11	4.77	1.14	4.75	3.77	4.87	1.42	4.43	1.39	4.39	3.28	4.38
jackson-core	2.12	5.35	2.28	5.34	4.11	5.43	2.01	6.13	2.03	6.06	4.61	6.17
jfreechart	1.15	3.74	0.98	3.73	3.44	6.43	1.38	3.26	1.07	3.25	3.11	5.73
joda-time	2.00	7.65	2.14	7.62	6.94	7.91	2.41	8.32	2.62	8.29	8.02	8.39
jsoup	1.85	8.72	1.70	8.71	8.67	17.47	2.20	7.36	2.00	7.32	7.11	15.46
mybatis-3	8.74	16.03	3.17	15.97	14.48	16.27	4.83	15.04	2.23	14.71	11.35	15.66
zxing	3.88	14.66	1.71	14.24	4.22	25.38	2.59	12.04	1.91	11.78	4.84	21.04
Mean	2.99	7.63	1.88	7.67	5.94	11.01	2.78	7.15	1.85	7.18	5.25	10.24
StDev	1.96	4.10	0.58	3.99	4.08	7.62	1.61	3.47	0.50	3.37	3.08	6.68

Table 6.14.: dl_{AVG} values for each project and each test set for the **ALL** and **DISJ** mutant data sets.

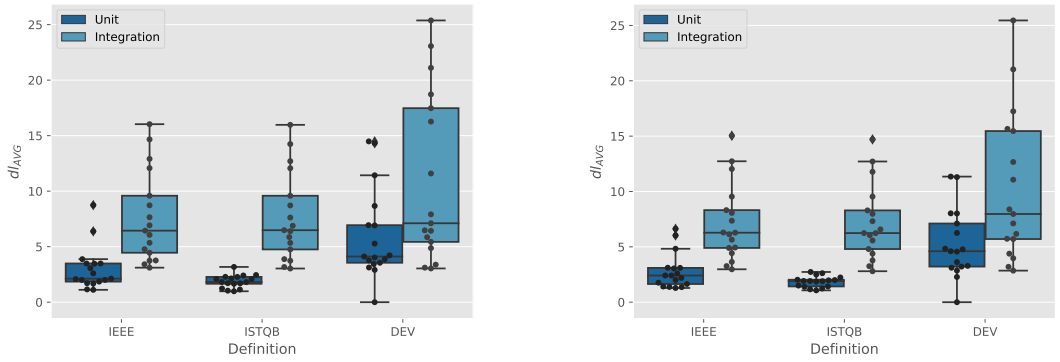


Figure 6.2.: Box-plot of the dl_{AVG} values for unit and integration tests and the IEEE and ISTQB definition, as well as the DEV classification for the **ALL** (left) and **DISJ** (right) mutant data set. The points in the plot represent the concrete values for each project.

$DL_{AVG}(U_{IEEE})$ and $DL_{AVG}(I_{IEEE})$, as well as the $DL_{AVG}(U_{DEV})$ and $DL_{AVG}(I_{DEV})$ sets have equal variances for both mutant data sets, as the Brown-Forsythe test shows. Therefore, we apply the Mann-Whitney-U test for the $DL_{AVG}(U_{IEEE})$ and $DL_{AVG}(I_{IEEE})$, as well as the $DL_{AVG}(U_{DEV})$ and $DL_{AVG}(I_{DEV})$ sets for both mutant data sets. The Welch t-test (Section 2.3.5.3) is applied on the $DL_{AVG}(U_{ISTQB})$ and $DL_{AVG}(I_{ISTQB})$ sets for both mutant data sets.

For the $DL_{AVG}(U_{IEEE})$ and $DL_{AVG}(I_{IEEE})$ and both mutant data sets the U-statistic is significant at the 0.005 alpha level. Moreover, the t-statistic is significant at the 0.005 alpha level for the $DL_{AVG}(U_{ISTQB})$ and $DL_{AVG}(I_{ISTQB})$ and both mutant data sets. Hence, we can conclude that there is a statistically significant difference in the defect-locality for IEEE and ISTQB unit and integration tests using both mutant data sets. However, for the $DL_{AVG}(U_{DEV})$ and $DL_{AVG}(I_{DEV})$ and both mutant data sets the U-statistic is not significant at the 0.005 alpha level.

In addition, we calculated Cohen's d for the sets, where we found significant differences. The effect size is "very large" for the IEEE definition and both mutant data sets ($d = 1.4437$ (**ALL**) and $d = 1.6160$ (**DISJ**)), while it is "huge" for the ISTQB definition and both mutant data sets ($d = 2.0339$ (**ALL**) and $d = 2.2046$ (**DISJ**)).

Answer to RQ 2.3: Table 6.14, as well as Figure 6.2 indicate that there are differences in the median and the mean between unit and integration tests for both definitions, as well as the developer classification, and regardless of the mutant data set being used. Our statistical tests (including "very large" and "huge" effect sizes) illustrate that there is indeed a difference between unit and integration tests regarding their average defect-locality for both definitions. However, the differences in the defect-locality between

unit and integration tests for the developer classification are not significant. For the IEEE and ISTQB definition, we can confirm that unit tests have (on average) a lower defect-locality than integration tests and therefore the pinpointing of bugs is easier, as developers do not need to go through a very big call stack. This difference can not be confirmed for the developer classification.

7. Qualitative Evaluation of the Differences between Unit and Integration Tests

We assessed relevant research literature, developer comments, and further information to extract indications regarding the question if the differences between unit and integration tests are still valid nowadays. We evaluated the resources that we found (e.g., blog posts that highlight developer opinions) to gain an understanding of the research and industrial landscape (i.e., the research and practical view) and to create a holistic view on the RQs in this thesis.

Within the following sections, we present the results of our qualitative evaluation. Section 7.1 summarizes our results regarding the differences in the test execution automation between unit and integration tests. The results regarding the different test objectives are presented in Section 7.2. In Section 7.3, we summarize the results regarding the differences in the costs of unit and integration tests.

7.1. Evaluation of RQ 2.4: Test Execution Automation

Test automation is a broad field, including the generation of tests, automatic prioritization of tests, as well as the automatic test selection. Within this section we focus on the test execution automation. Test execution automation tools provide means for the automatic execution and result comparison of tests. Hence, the expected result is automatically compared to the results calculated by the test object and reported back to the developer.

The standard literature points out that unit tests should be automated [11] and are indeed often automated [15]. However, we could not find any statements in the standard literature regarding the test execution automation of integration tests. Within this section, we present the research view regarding the test execution automation of unit and integration tests in Section 7.1.1. The practical view on this question is presented in Section 7.1.2. Finally, in Section 7.1.3, we summarize our results and give an answer to the RQ if there is a difference in the test execution automation between unit and integration tests.

7.1.1. Scientific View

The research literature that we collected agree on the fact that the execution of both, unit tests as well as integration tests, can be automated [292]. This is highlighted by the large number of works in the different areas that are connected to the automatic execution of

those. For example, there are several publications that present new or improve existing unit testing frameworks (e.g., [293, 294, 295]), which make an automatic execution of unit tests possible or more efficient. Furthermore, there are several works that ports unit testing frameworks to different system types (e.g., agent systems [296] or embedded software [297]). Moreover, there are different published studies that have a closer look at how the automation of the execution of unit tests is used or improved within an industrial context (e.g., [297, 298, 299]).

This looks similar for the automation of integration testing. There are publications of frameworks that ease the execution of integration tests (e.g., [300]) and studies that evaluated the automation of integration testing in an industrial context (e.g., [301]). Furthermore, the automatic execution of integration tests is also used in other research areas. One example is the area of mutation testing. In the paper by Delamaro et al. [262] an integration level mutation testing approach was introduced. Here, the study in the paper integrated different mutations and executed the integration tests automatically through an integration testing framework that included the automatic execution of tests [262].

However, the literature also states that the automation of integration tests is more difficult than the automatization of unit tests [302]. On the other hand, there exist studies that show that the automatic execution of integration tests is beneficial for the project (e.g., [303, 304]). The difficulty of creating automated integration tests is highlighted by studies that evaluated the automatization of tests in an industrial context (e.g., [303]). These studies showed that most companies primarily automated the execution of unit tests. The reasons for this are manifold. One is that the business logic tested via integration tests, often needs other modules, e.g., a databases, to be available. If these modules exist only once, multiple parallel test executions might interfere with each other. Hence, the complexity of the automation of integration tests is higher than for tests on the unit level.

7.1.2. Practical View

Our evaluated resources to create a practical view agree on the fact that the automation of test executions is beneficial. They state that tests are important and a high number of tests and their executions increase the confidence in a software product. Due to the large number of tests it is often impossible to manually execute all of them. Therefore, an automated test execution is needed [305].

Furthermore, the evaluated resources agree that both (integration and unit tests) should be automated. During our study we found several tools that automated the execution of unit and/or integration tests (e.g., [172, 306, 307, 308, 309]). They mostly differ in the focus and supported programming language(s). However, the automation of unit and integration tests is done via two different approaches. The literature states that unit tests are mostly executed locally in the environment of the developers [310, 311]. Furthermore, test execution frameworks are often integrated into Integrated Development Environments (IDEs) (e.g., [312, 313]) so that the execution of the tests can be done from the IDE of the devel-

oper and the results are presented within the IDE. However, the way that integration tests are executed is different, as they are (mostly) not executed by the developers, but through a CI system [305, 314]. For that, the execution of the tests is integrated into the build process (e.g., via the use of specific plugins for unit [119] and integration [120] tests) of the project and at certain points in time (e.g., after each change or before a release) the project gets build on a server which also executes all unit and/or integration tests. This separation is often done because integration tests take longer to execute [314]. However, while these two different approaches are also used by the projects that we analyzed within our quantitative analysis, we could not make out a separation of test types based on these approaches. Instead, all of the projects that we analyzed executed all their tests (i.e., unit and integration tests) locally and in the CI system.

7.1.3. Summary

We found indications that the automated test execution of integration tests is possible and is done in an industrial context. However, it seems that it is not as comprehensively used as the automated execution of unit tests. The reasons for this are the higher complexity of integration tests. On the other hand, our analysis of the practical view provides indications, that unit and integration tests are both executed automatically. However, different approaches are used to automate the execution of tests on those levels. While unit tests are mostly executed by developers through integrations of test execution frameworks into IDEs, integration tests are mostly executed automatically via a CI system.

Answer to RQ 2.4: The analysis of the research and practical view provide us indications that suggest that unit and integration tests can both be executed automatically. Our analysis highlights, that the automatic execution of unit and integration tests is part of active research. In addition, we found indications that both test types are automatically executed within industry. However, scientific studies show that the automatic execution of integration tests is not as common as the automatic execution of unit tests.

7.2. Evaluation of RQ 2.5: Test Objective

Some literature (e.g., [9]) state that the efficiency, maintainability, and robustness of software should be tested by unit tests on the unit level instead of doing this on the integration or system level. Within this section, we focus on this statement and evaluate separately for efficiency (Section 7.2.1), maintainability (Section 7.2.2), and robustness (Section 7.2.3) if this is really the case.

7.2.1. Efficiency Testing

Efficiency testing is used to test the usage of resources (e.g., CPU, RAM, network) of the SUT/component and is often measured via certain characteristics like response time or memory usage [9]. There exist several types of performance testing, including load testing (tests how a system behaves under a specific expected load), stress testing (tests how a system behaves under extreme load), and spike testing (tests how a system behaves under sudden load spikes). Within this section, we present the results of our analysis of the research (Section 7.2.1.1) and practical (Section 7.2.1.2) view on the topic of efficiency testing.

7.2.1.1. Scientific View

The field of efficiency testing is rather broad. There are numerous papers that describe the results of case studies or give experience reports on the application of efficiency testing (e.g., [315, 316, 317]). Furthermore, there exist several papers that propose efficiency testing approaches, e.g., to automatically generate efficiency testing test suites (e.g., [316, 318, 319]) or that ease the analysis of performance data [320, 321]. Our analysis is focused on paper that describe how efficiency testing was applied to a software and on which level.

One of the first paper that described an approach for efficiency testing was authored by Avritzer et al. [322]. Within their paper the authors present three automatic test case generation algorithms to test the allocation mechanism of specialized software (i.e., telecommunications software systems). They used these algorithms to perform efficiency testing for several real industrial software systems.

Another paper by Avritzer et al. [323] presents an approach to compare the efficiency of an existing platform and an architecture that should work as a replacement to evaluate if the new architecture is capable of handling the workload (i.e., if the resources are used efficiently). Within their approach, they did not port the software to a new platform to do the efficiency testing, but artificially created workloads to test the new and old architecture.

Another relevant paper was published by Weyuker et al. [315]. They report within their paper that very little work was published in the field of efficiency testing. The authors present an approach to software efficiency testing together with a case study on a large industrial project where the approach is evaluated. Their approach includes the "design of test case selection or generation strategies specifically intended to test for performance criteria rather than functional correctness criteria" [315].

More recently, Zhang et al. [316] proposed an approach to automatically generate test cases for the efficiency testing of multimedia systems. Their approach makes use of constraint solving techniques to construct test cases that target the resource saturation point (i.e, the point when all resources are in use) in multimedia systems. They implemented their approach into several tools that were applied to real industrial projects.

Garousi et al. [324] proposed an approach to assist in the generation of test cases, which are focused on revealing efficiency related defects using Unified Modeling Language (UML) models. The UML model of a SUT is used as an input for their approach. Afterwards, a test model is build to enable the subsequent automation steps. This model, together with some test parameters (i.e., the objectives, set by the user), are used by an optimization algorithm to derive the efficiency test requirements. These can then be used to define the test cases. However, as with the other approaches, the SUT in this case is the whole system.

All these papers do efficiency testing on system level. We were not able to find a paper that did efficiency testing on another level. This shows that efficiency testing in the context of research is done on system level.

7.2.1.2. Practical View

There exist numerous companies that provide a service to do efficiency testing on applications (e.g., [325, 326, 327, 328]), even for modern technologies like Virtual Reality (VR) [329].

There are also several tools that can be used to do efficiency testing on applications (e.g., [330, 331, 332, 333]). They mostly differ in their application domain (e.g., load testing of websites, application servers, or APIs) [334]. Furthermore, there exist tools for manual efficiency testing (e.g., JVMMonitor [335], VisualVM [336]). Those tools are profilers that measure, e.g., the usage of RAM, CPU, as well as the response times of units and functions. These tools do not provide means to create or generate test cases.

That efficiency testing is an important topic for practitioners is also highlighted by several blog posts that we found within our analysis. There is one guide by Microsoft [337] that provides a step-by-step tutorial to perform efficiency testing of web applications. This guide only describes how efficiency testing can be done on the system level. Furthermore, there are several other blog posts by developers (e.g., [338, 339, 340]) that describe different approaches to do efficiency testing of applications or websites. While they differ in their approach, used tools, and application domain, they all agree on the level on which efficiency testing is applied, i.e., the system level. One exception is the blog post by Stackify [341], which highlights that efficiency testing is mostly done on the system level, but "There is value in testing individual units or modules." [341].

7.2.2. Maintainability Testing

Maintainability testing is used to test all characteristics that influence the difficulty to change a program. These characteristics include code structure, modularity, code comment quality, and so on [9]. However, most of those characteristics can only be evaluated by static tests (i.e., the source code is not executed). One technique that is especially useful for testing the maintainability are reviews [9].

Maintainability is a concept that is rather broad. The ISO 25000 standard characterizes maintainability by dividing it into five different categories [342].

- **Modularity:** degree to which a software is separated into different components so that a change to one component does not (or only on a minimal level) affect another component.
- **Reusability:** degree to which code parts can be reused in other components.
- **Analysability:** degree to which it is possible to trace the impact of a change to parts of the system, or to detect the causes for software defects, or to find parts that should be modified.
- **Modifiability:** degree to which a system can be modified without introducing defects or reduce the systems quality.
- **Testability:** degree to which it is possible to fulfill test criteria for a system and to which tests can be executed to measure if those criteria are met.

Within this section, we present the results of our analysis of the research (Section 7.2.2.1) and practical (Section 7.2.2.2) view on the topic of maintainability testing.

7.2.2.1. Scientific View

One of the earliest publications on maintainability testing was written by Oman et al. [343]. They tried to determine different factors that have an influence on the maintainability of software. They present different metrics that can be used to measure these factors. Oman et al. [343] discuss the reasoning for these metrics and how they fit to the determined factors. Furthermore, they present an approach to compile these metrics into one metric, which should measure the maintainability of a software system. This metric is called the Maintainability Index (MI).

Within their follow up publication [344] the authors makes use of the MI. They performed a case study on 8 software systems to calibrate the MI (i.e., defining factors of the chosen metrics). Afterwards, they evaluated their MI formula on 14 industrial projects by comparing the output of the formula with expert opinions. The evaluation highlighted that there is a correlation between the MI and the expert opinions. Hence, they concluded that the MI is a useful measure for the maintainability of software.

That the MI is a useful metric to test the maintainability of software is also highlighted by other publications (e.g., [345, 346, 347, 348]). Based on these publications, there are several others that try, e.g., to improve/extend the maintainability index (as a recent review highlights [349]) or that develop new models to grasp the concept of maintainability and make it testable (e.g., [350, 351, 352]). There are even different forms of MIs, e.g., [353, 354]. Additionally, the research community defined several other metrics that are connected to the maintainability of software like the readability [355].

The testing of maintainability itself (e.g., through the calculation of the MI) is mostly done through the calculation of source code metrics (e.g., Cyclomatic Complexity (CC)) and evaluating if a certain threshold is exceeded. These metrics are calculated on the unit level and aggregated later on. However, these metrics can also be used to test the maintainability of single units (e.g., by defining a threshold for the unit level).

Spillner et al. [9] describe, that the maintainability should be tested via reviews on the unit level. Extensive research is also done in this direction to improve the quality of reviews or make the reviewing process more efficient. There exist approaches to recommend reviewers for code parts (e.g., [356, 357]) or to improve the overall review process (e.g., [358]). It is possible to perform code reviews on the unit level. On which level reviews are performed (i.e., unit, integration, or system level) is highly dependent on the employed reviewing process and the system under review. However, most code reviews are done after a change of a developer, i.e., one developer reviews the change of another one and, therefore, several code units may be affected by the review.

7.2.2.2. Practical View

The interest within the developer community on maintainability testing seems to be limited. We found more resources discussing the topic of maintainable tests (e.g., [359, 360]) than maintainability testing. However, there exist several resources that are focusing on this topic.

One resource is a blog post by Nupul Kukreja [361]. Within his post he defines the term maintainability and its sub-characteristics. In addition, Kukreja shows different tables to each of these sub-characteristics, which include different code quality/maintainability metrics together with their correlation to quality, importance, feasibility of automated evaluation, ease of automated evaluation, completeness of automated evaluation, and units. He makes clear that the maintainability of software code is very complex and therefore hard to assess via different metrics. However, there exist metrics with which it is possible to assess several aspects of maintainability. These metrics are not only unit-level metrics (e.g., CC or unit length), but also metrics that are calculated over all units (e.g., coupling, cohesion). Furthermore, Kukreja also highlights the importance of the MI, which we also described within the scientific view (Section 7.2.2.1).

During our analysis we found several static source code analysis tools that are able to calculate (parts of) the metrics that are mentioned by Kurkeja and that are used to calculate the MI (e.g., [354, 362, 363]). Two of these tools are SonarQube [362] and Checkstyle [363].

SonarQube [362] is a platform that is able to statically analyze source code to assess its quality. The results of this analysis are then presented within a website that can be browsed by developers. It calculates metrics like the number of duplicated lines of code, the complexity of code, or the comment lines of code. Furthermore, SonarQube can calculate metrics that are connected to the maintainability, e.g., it is able to calculate a “Maintainability Rating”, which is a rating that is based on the ratio of the code size and the estimated

time to fix all maintainability issues of the code. All these maintainability related metrics that are calculated by SonarQube are calculated for the whole system and not for the units themselves.

Checkstyle [363] is a static source code analysis tool that is able to check the adherence of code to given style guidelines. For example, it can determine if the brackets of loops are at the correct place (i.e., in accordance to the style guideline given to Checkstyle). Through this, Checkstyle can contribute to assessing the maintainability of source code, but it cannot assess all of its facets.

Another technique that is often discussed for maintainability testing is code review. There exist several platforms that can assist developers with the execution of them, e.g., CodeFlow [364], GitHub [180], or Gerrit [365]. All of these tools and platforms have in common that changes that are committed to the VCS are reviewed. While it is possible to review only changes for one unit, this is often not the case as several units are changed within one commit. Therefore, most of these reviews can

7.2.3. Robustness Testing

Robustness is defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [27]. Hence, the goal of robustness testing is the development of test cases and environments where the robustness of a system can be assessed. Spillner et al. [9] argue that the testing for robustness should be done by unit tests (i.e., negative tests). Within this section, we present the results of our qualitative analysis regarding the test level of robustness testing. Section 7.2.3.1 presents the results from the scientific view, while Section 7.2.3.2 highlights the practical view on robustness testing.

7.2.3.1. Scientific View

Over the years, several tools and approaches were developed to assess the robustness of a system or component. One of the earliest works in the field was done in the early 1990s. Several approaches were developed that injected physical faults into the environment of the software (or hardware) without damaging the tested component. There are several tools that simulate physical faults by injecting defects through the software, e.g., FIAT [366], FTAPE [367], or FERRARI [368]. FIAT [366] uses probes that are manually placed to change the binary process image in memory, while the program is executing. FERRARI [368] uses a similar approach, but instead of the manually integrated probes, it uses software traps to allow the emulation of hardware faults. FTAPE [367], on the other hand, generates random workload that is processed by the software and during this process it injects faults into the system. All of these techniques work on system level.

Another robustness testing technique, which is still used today in security related testing [369] is the generation of random inputs. Random inputs are generated and given as

parameters to functions or systems to evaluate how they react on those. A robustness defect was found, if the tested component or system crashes or does not respond. These techniques are rather efficient, as the creation of random inputs is not expensive in terms of computational power [370]. Those techniques are used in several studies (e.g., [371, 372]) and are also effective in testing the robustness of modern standard software [372]. One tool that implements this concept is Fuzz [371]. Fuzz [371] is a tool that generates random characters which can then be used as an input for command-line tools. While random input testing can be applied on any testing level, the Fuzz tool is only capable of testing command-line tools and therefore performs tests on the system level.

The generation of invalid inputs can also be done in a more structure way. These techniques use inputs that are inside and outside of the allowed domain and specifically select inputs by defined rules [370]. These rules can be generated, e.g., grammar-based [373] or specified manually for specific domains [374]. Ghosh et al. [373] present a tool called RIDDLE, which uses the input grammar of the SUT to generate test cases. The input that is used by these test cases conforms to the basic syntax of the SUT's input. However, RIDDLE is not designed to test single functions, but the whole software system. On the other hand, Mendonca et al. [374] present a robustness testing study done on the Windows device driver toolkit. Within their approach, they determine the functions that are used within device drivers and select the ones that are used most often. For these functions, workload (i.e., input) is generated according to given rules to evaluate if the function fails. Here, the testing is done on unit level and not on system level.

The type-specific approach to test the robustness of software was introduced in the early 2000s. It uses the random generation of inputs, as well as the invalid input generation technique explained above. Within type-specific testing "valid and invalid inputs are defined for the data types used in the system's interface functions, and the robustness tests are generated by combining the values for the different parameters" [370]. One approach and tool that is popular within this area is Ballista [375, 376]. The robustness testing methodology developed by Koopman et al. [375] creates test cases for a single pair of test values for single functions. It parses the interface definition of the functions to be able to create fitting test inputs for them. Hence, Ballista focuses on single functions instead of the whole system.

An extension to the generation of type-specific tests is the generation of object-oriented tests. With the increasing popularity of object-oriented software this robustness testing technique gained a lot of interest. The advantage of object-oriented systems is that within those systems a parameter graph containing the type structure of the parameters can be built. This graph describes how specific object types can be generated by calling constructors and/or public methods of classes within the software system. Using this information, it is possible to easily generate invalid objects by tracing back the different calls in the parameter graph [370]. One popular tool that implements these techniques is the JCrasher tool [377]. It "examines the type information of a set of Java classes and constructs code fragments that will create instances of different types to test the behavior of public methods under random data" [377]. Hence, it is a tool that works on the unit level.

More recently, due to the increasing popularity of model-driven development, model-based robustness testing approaches were developed. Models that are created during the development can be used to generate robustness test cases, e.g., "by looking for extreme values and conditions on the basis of the pre- and post-conditions, invariants, and constraints fixed in the design model" [370]. An example is IOLTS [378], which is an approach that uses path searching and model mutation techniques. This approach is applied on the system level. Another framework was introduced by Oláh et al. [379]. This framework is directly integrated into the Eclipse IDE. It uses UML class diagrams to inject faults into the software and monitors its execution. The framework provides means to configure the fault injection process and the robustness testing experiment overall.The tester needs to provide a workload that is given as input to the application, i.e., none of the above mentioned input generation techniques are applied. Hence, this framework works on system level, as the whole system needs to be executed with the provided workload.

Most recently, robustness testing techniques have been used in several specific application domains, e.g., user interfaces (e.g., [380]), high availability middleware (e.g., [381]), real-time executives like microkernels (e.g., [382, 383, 384]), online transaction processing systems and database management systems (e.g., [385, 386]), and web services (e.g., [387, 388, 389]). All of these approaches have in common that they are performed on the system level, e.g., by sending messages to the systems's exposed API [387].

Another specialization of robustness testing that gained a lot of interest is the field of penetration testing. In penetration testing the execution of the system is monitored, while (possibly) malicious inputs are given. The goal of penetration testing is the detection of potential vulnerabilities in the software [370]. There exist several categories of approaches [370]. The black-box penetration testing approaches test the robustness of a system on system level, by providing inputs to the system interface (e.g., [390, 391, 392]). The gray-box penetration testing approaches complement the black-box approach by integrating white-box techniques. These techniques (e.g., [393, 394]) analyze or observe the execution of the program, while the input is provided through the system's interface.

7.2.3.2. Practical View

While robustness testing is an important topic in research that was continuously advanced during the years, we only found a limited amount of resources that highlights the practical view on robustness testing. A website [395] entry explains the term robustness testing, its significance, and the challenges, but besides this website, few resources were found.

One source is the software robustness tutorial [396], which was held by Vincent Sinclair employed at Nokia. Within his tutorial, he explained the term robustness and robustness testing and explains where robustness testing is (or should be) applied. He highlighted that robustness testing is an activity that should be applied on all test levels and areas, including unit, integration, and system level, as robustness defects might not be detected otherwise. On the unit level, several techniques are used, e.g., the provisioning of invalid inputs to

functions, or performing boundary tests (i.e., testing the boundaries for a given function parameter). Techniques on the integration level include input overload/storms to see how the system can handle it. On the system level, different feature interactions are tested, as well as the user interface. Sinclair [396] highlights in his presentation that robustness tests for all levels should be included in the software robustness test plan.

Another source is the white paper published by Bridgewater et al. [397]. The authors describe their approach to improve the robustness testing at VMware. They present two different pseudorandom test generators, which are able to create test cases using knowledge extracted from the SUT. This knowledge is used to select input data that is most likely to produce relevant test cases from a wide range of values. Their approach is able to create complex test conditions, which could not be covered by unit tests alone. Bridgewater et al. [397] did several experiments with their approaches and found that the presented generators "have been critical in finding problems in our complex software and are a good complement to other test methods such as directed unit tests and running real-world system-level workloads." [397].

Besides these resources mentioned above, we found several tools that are used by practitioners to test the robustness of software. We found one commercial product that is able to test the robustness of C/C++ software [398]. However, it is not explained in detail how this tool works or how test cases are generated. Another framework that is often used in terms of robustness testing is JUnit for Java applications [172]. JUnit can be used to execute test cases that evaluate the exception handling procedures of a software by asserting if the correct exceptions are thrown. The test cases can be on unit, integration, or system level. Other tools that we found were presented by Bahmutov [399]. Within his blog posts he explains his tools that could help in robustness testing by mocking responses (e.g., by slowing down requests, returning specific status code, or returning specific mock data). These tools work on the system level.

7.2.4. Summary

During our analysis of the scientific view regarding the differences between unit and integration tests in context of efficiency testing, we found indications that the use of efficiency testing tools is well established at the system level. We could not find any evidence or research that tried to perform efficiency testing on unit or integration level. Woodside et al. [400] mentions several reasons for this imbalance, e.g., a lack of theoretical justification for the methods that try to improve the efficiency (i.e., measurement data is provided, but the interpretation is still open) or the difficulty to correlate the events from different systems within distributed systems [400]. The analysis of the practical view provided us insights into two approaches for efficiency testing that are used by practitioners. One approach is done on the system level, where the user behavior is mimicked while the resource usage of the system is monitored. The other approach is the manual efficiency testing on smaller

levels by using profilers and running existing test cases (possibly enhanced by profiling output [401]), whereas the level targeted by these test cases is not specified.

Research tries to realize the concept of maintainability (e.g, through the development of new metrics) and support the maintainability testing process (e.g., through the support of code reviews). However, for the maintainability to be thoroughly tested (i.e., to test every aspect of maintainability), it is necessary that the calculation of metrics like the MI or the performing of code reviews span all three testing levels (i.e., unit, integration, and system level). On the practical side, our results indicate that maintainability testing is a topic that seems not to be present within the development community, as the amount of found resources is rather low in comparison to other topics that we evaluated. The maintainability of tests seems to be a topic which is more relevant. While tests are important for the maintainability of a product, they are not used to test the maintainability itself. However, the resources that we found agree that maintainability testing is a complex field, which is mostly tackled by the calculation of metrics that are connected to the maintainability, or the execution of code reviews.

Within our analysis of the scientific view regarding the differences between unit and integration tests in the context of robustness testing we found indications, that extensive research is done in this field. Several tools, techniques, and frameworks were developed throughout the years. These tools and techniques can be used as an internal step in the development process, e.g., by applying the robustness testing early on and develop unit tests for it. On the other hand, robustness testing can also be executed after the release of the system on the system level. During our analysis of the practical view, we found indications that practitioners know of the importance of robustness testing. Nevertheless, from the resources it is clear that robustness testing can be done on all test levels and should be done on all test levels, as focusing robustness testing on only unit level would not be sufficient to test for complex test conditions.

Answer to RQ 2.5: For efficiency testing the research and the practical view agree: within both view we found indications that show that efficiency testing is mostly done on system level, while there is a need for automated efficiency testing on lower levels (e.g., unit or integration level). Currently several manual approaches are in use to test the efficiency on lower levels, e.g., through the use of profilers.

In the field of maintainability testing, extensive research is done to develop approaches to automate the process of maintainability testing and to develop metrics that can describe the maintainability of software. One of such metrics is the MI which is used in research and practice, as our results highlight. The MI can be calculated on several levels, e.g., unit, integration, or system level. Another technique that is often used to test the maintainability of a software is code reviews. Within our analysis, we found indications that those reviews are often done change-based and, therefore, not only on unit level. As the ISO 25000 standard [342] highlights, the maintainability of

a software should be tested on different levels, as different aspects of maintainability like the modularity or analysability, can not be tested on one testing level alone.

There exist an extensive body of knowledge on robustness testing. Currently, a lot of research is focused on this area, as our results highlight. Robustness testing can be done on unit, integration, and system level. That robustness testing is an activity that should be done on all test levels is highlighted by the results of our analysis of the practical view.

7.3. Evaluation of RQ 2.6: Test Costs

Some literature state, that the development and execution costs of unit tests are smaller than the ones for integration tests (e.g., [13]). Hence, within this chapter, we analyze the scientific view (Section 7.3.1) and practical view (Section 7.3.2) on this topic, while focusing on the differences in the costs between unit and integration tests. Finally, in Section 7.3.3 we answer our RQ regarding the differences in the costs between unit and integration tests.

7.3.1. Scientific View

There exist a lot of different cost models that try to estimate and predict the costs of software testing in general, e.g., by Berry Boehm [402]. However, there exist very few empirical evidence regarding the costs of software testing on the different test levels from real software projects, as a systematic literature review highlights [403]. One potential reason is that researchers often do not have access to such data from industrial projects. This kind of data is often highly confidential and therefore not shared easily. Furthermore, such data is not available for open-source projects, as these projects are only developed by contributors (that do not get paid by the project[11]) and the development process is different from a formal one that might get applied in industrial projects [146]. Most papers that focus on the costs of different test levels or techniques perform experiments using humans (e.g., [404, 405, 406]). These studies are often not done in a realistic context and are often small in their size (i.e., number of students/developers and size of program that is tested). Within this section, we focus on studies that were done on real industrial software projects and that report the costs for either unit testing or integration testing.

A paper by Williams et al. [299] evaluated the effectiveness of unit test automation at Microsoft. The authors performed a post-hoc data analysis using the code, test, bug, and other repositories from a Microsoft team, which changed their testing practices from ad hoc and individual testing practices to unit testing practices. In addition, Williams et al. [299] executed a survey of developers and testers from this team after their transition. Afterwards, four developers and testers were interviewed and asked questions regarding the efficiency and applicability of the testing practices. Williams et al. [299] found, besides other results,

[11] Recently, diverse large companies put resources into the extension and development of open-source projects.

that the writing of unit tests increases the development time by 30%. Hence, the writing of unit tests increases the costs of the development process. However, the they also found a "20.9% decrease in test defects" [299].

Another study that had a closer look at the cost effectiveness of unit testing was done by Delgado et al. [407]. They performed a case study within a financial institution. Within their study they first chose fitting software projects, implemented unit tests for it, identified prevented defects, and performed a cost and savings analysis. As one of the few studies, they report the costs and savings not only in hours spend, but also in U.S. dollars. Within their case study the developers spend 323 hours (totaling in 8,721$ cost) in developing unit tests and saved 73 hours (totaling in 2,462$ cost). The savings were derived from the number of defects in production and testing that were found by unit testing. Hence, one major finding of their case study is "that unit testing has a high cost in terms of time (effort) and money)" [407]. Furthermore, they state that 28% of the total development time was dedicated to the development of unit tests and therefore the effort of unit testing is about 4.4 times greater than the return. The numbers are slightly different for the investment in money, which is 3.5 times greater than the return. The application of unit testing was not profitable in their case study.

Besides these two studies that focus on assessing the cost of unit testing within an industrial context, we did not find any papers that do the same for integration testing. However, there exist two papers which report differences in the costs of unit and integration testing.

The first was written by Ellims et al. [16]. The authors collected data from three different industrial software projects that performed unit testing. The three safety-related projects are between 3500 LOC and 13500 LOC and were developed by the same company. Ellims et al. [16] accumulated the data from different data sources (i.e., timesheets, review records, change requests, code coverage), to analyze the economics of unit testing. One finding is that "unit testing can be between 2 and 13 times more effective than other test activities applied" [16][12], if the data that was collected from one of the projects is representative.

The second paper that provides a case study on a safety-critical application was done by King et al. [408]. During the development of this application, they recorded different metrics (e.g., the number of found faults at different testing stages), which they then report within their paper. While the main focus of the paper is the comparison of proof to the more traditional types of testing, we can extract several information from the paper. In their analysis they compared proof with various types of testing, including unit and integration testing, in terms of their efficiency. Overall they found, that proof is more efficient than the traditional testing techniques. However, more important for our context, they also found that while unit tests detect more defects within the software than integration tests, integration test are more efficient in terms of number of faults found per day of effort. Hence, the effort (i.e., the costs) of unit testing is 25 times higher than for integration testing [408].

[12]Effective in this case means the spent man hours per detected defect.

7.3.2. Practical View

Our search for the practical view on this topic revealed a uniform opinion regarding the differences of cost between unit and integration tests with one exception. Most of the resource we found agree on the point that integration tests are more costly (in terms of writing and running the tests) [32, 29, 409, 410, 411]. However, we did not find any real empirical studies or data on this topic, but experiences of developers. Nevertheless, developers made clear in their articles, that empirical data on this topic would be highly appreciated [32].

The one single exception that we found separates the costs of unit and integration testing based on the project type [412]. Kudryashov distinguishes between projects that need to be integrated into an existing environment and with existing systems (i.e., brownfield projects) and those which do not need to be integrated (i.e., greenfield projects). He writes, that in his experience integration tests are more costly in the maintenance than unit tests. This holds true for both project types. However, the introduction of new unit tests is more costly than the introduction of integration tests for brownfield projects only, while it is nearly equal for both testing types for greenfield projects.

7.3.3. Summary

Within our analysis of the scientific view, we found statements that unit tests are cheaper than integration tests and statements that integration tests are cheaper than unit tests. While it is clear and uncontroversial that unit testing has its costs (as has integration testing), research seems to be discordant in this respect. While the paper by Ellims et al. [16] state that unit testing was more effective (i.e., less costs per found defect) than other testing techniques, King et al. [408] comes to the conclusion that integration tests are more effective than unit tests (and have a higher effort in developing). On the other hand, the resources that we found to enlighten the practical view on this topic agree on the fact that integration tests are more costly (in terms of developing and running the test) than unit tests.

Answer to RQ 2.6: Overall, our results showed that the current research on the topic of costs of tests on different levels cannot provide a definite answer, as we found indications in both directions. However, our analysis of the practical view show that in the experience of the developers integration tests are (mostly) more costly than unit tests. Moreover, we found that we are missing empirical studies with real software projects on this topic to come to a definite conclusion.

8. Discussion

Analyzing the results, we see that most of them do not represent what we have learned or what we teach teach at the university, or what is defined in the literature. Hence, the results of this thesis are interesting from several perspectives. Within this Section, we discuss our results and draw conclusions from the perspective of education and academia (Section 8.1), as well as from a practical perspective (Section 8.2). Moreover, in Section 8.3 we explain the threats to validity for our study, as well as our validation procedures that we performed to assess and/or lower the threats.

8.1. Education and Academia

Academia, as well as organizations like the ISTQB, teach that developers should follow the concept of the testing pyramid: they should develop more unit than integration tests, as unit tests are the basis on which the integration tests are built upon. However, our results show, that most of the projects do not follow this concept anymore and instead follow the proposal mentioned in the introduction, i.e., that there are less unit than integration tests developed (Section 5.2.2). While there is a difference in the number of unit and integration tests depending on the used definition (i.e., IEEE or ISTQB), this result holds true for both of them. Nevertheless, our results also highlight that the developer classification of tests and the classification according to the definitions are different (Section 5.3.2 and Section 5.4.2). This result suggests, that we need a better education for developers and make clear how unit and/or integration tests are defined or that the definitions must be adapted to modern development contexts.

With our quantitative analysis, we did not find any difference in the execution time between unit and integration tests, if we assess the execution time per covered line of code (Section 5.2.2). The execution time is often neglectable nowadays, due to the research on and development of, e.g., CI systems which automate the execution of long running tests. Moreover, the computational power that most companies have at their disposal contribute to the reduction of the execution time of tests.

All of our assessed literature agree, that unit and integration tests detect different types of defects and that a separation into unit and integration tests is important. However, our quantitative results show that there is no statistically significant difference in the effectiveness between unit and integration tests (Section 6.2.3.1). More interestingly, we could not find any evidence for the statement that unit and integration tests find different types of defects

(Section 6.2.3.2). Our results on the defect-locality showed that the common belief that unit tests are better able to pinpoint the source of a defect than integration tests holds true. We found that the defect-locality metric is significantly lower for unit tests than for integration tests for both definitions (i.e., IEEE and ISTQB). However, if we reuse the developer classification of tests, we can not find a statistically significant difference in the defect-locality between unit and integration tests.

Our qualitative analysis found several shortcomings in the current research on unit and integration tests. It shows that it is favorable to do efficiency testing on all testing levels, as performance problems might be detected too late or not at all. We also found that there is currently a need for the automated performance testing on lower test levels, i.e., there exist a research gap (Section 7.2.1). Furthermore, our results highlight that maintainability testing should and must be done on all test levels, if all aspects of maintainability should be assessed. There is a lack of automation besides the calculation of the MI. We are missing tools (and their evaluation) that are able to automatically assess and improve the maintainability of a software system considering all maintainability aspects (Section 7.2.2). Another research gap that we identified is the evaluation of costs of tests. There is a very limited amount of studies of a limited scope that assessed the costs of tests on different test levels (Section 7.3). Here, we need more research with bigger case studies using real world software systems.

Nevertheless, we also identified research areas which have substantially improved over the years. The area of automatic test execution (Section 7.1) or robustness testing (Section 7.2.3) are both research areas, where a lot of research was done. Especially the area of automatic test execution delivered usable results, e.g., software testing frameworks like JUnit [172]. There is still some research in these areas missing, e.g., to automate robustness testing on the integration level.

Overall, our results raise the question, if academia should develop another distinction criterion to differentiate between unit and integration tests. While we found that some differences exist (e.g., the defect-locality), most of the exercised differences are not statistically significant. Especially the most mentioned and therefore most important difference of unit and integration tests (i.e., that unit and integration tests detect different types of defects) can be identified as a software testing myth. Hence, it might not be a good idea to distinguish tests based on their structural content or which units they assess. Ammann and Offutt [1] reason that “most of the literature emphasizes these [test] levels in terms of when they are applied, a more important distinction is on the types of faults that we are looking for.” [1]. Hence, they propose to divide the tests on the test levels based on the defects they target. This thesis provides evidence that it might be beneficial to find another distinction criterion, e.g., by following the proposal of Ammann and Offutt [1]. Hence, research needs to be done to revise current valid definitions of unit and integration tests that we teach in academia, practice, and industrial certifications.

8.2. Practice

Software development and software testing goes hand in hand. The developed parts should always be accompanied by tests that exercise them. However, it is often hard to tell from a developer point of view, which test type might be favorable in certain situations. Our results can give hints to developers on which test level they should develop tests, if they pursue a specific goal.

Our results show a discrepancy between the developer classification of a test (i.e., unit or integration test) and the classification using common definitions (Section 5.4.2). This result has two interpretations. Either, it shows that we need a better education for developers in the field of software testing, or it highlights that we need modern definitions that better fit to the developer reality. The current definitions of the IEEE and ISTQB are historically grown and decades old. Through the growing technologization and the advances in the field of software engineering and software testing, the current definitions might do not fit to our modern software engineering world anymore. Modern practices like CI, DevOps, and new software testing frameworks contribute to this shift. However, more research is needed in this field to evaluate, if we really need new definitions and how they should look like. Nevertheless, this thesis highlights that the research community needs to act and work together with practitioners to solve this problem.

Moreover, our results show that it does not matter what kind of test (i.e., unit or integration test) a developer is creating, as we could not find significant differences between unit and integration tests for most of the differences that we evaluated. These results are also robust along the used definitions (i.e. the IEEE and ISTQB definitions). However, it also depends on the goal that the developer pursues with a test, if a unit or integration test might be favorable. For the test execution time we did not find any difference between unit and integration tests, if we assess the execution time per covered line of code (Section 6.1.3). If the test effectiveness is of interest, it also does not matter if a unit or integration test is created, as they both do not differ from each other in this aspect (Section 6.2.3.1). Interestingly, even if a developer targets specific defect types (e.g., interface defects), our results highlight that it does not matter if a unit or integration test is used (Section 6.2.3.2). This also holds true, if the robustness of a software should be tested, as our qualitative analysis highlights (Section 7.2.3).

While we can give hints to developers or managers on the usage of unit and integration tests, we cannot give concrete guidelines. During our qualitative analysis we found several shortcomings in our current knowledge on unit and integration testing. We cannot make conclusions about the costs of these two test types, as we only found anecdotes or experience report with our qualitative analysis and only very few and limited scientific papers on this topic. This is also valid for other aspects like, e.g., the different test objectives. Here, research is missing to come to a definite conclusion and to create guidelines for developers to support them in choosing the best test in certain circumstances.

To come back to the question from our introduction, where we presented the proposal of

developers to reduce the amount of unit tests and develop more integration tests instead: as explained above, it depends on the goal one wants to achieve to evaluate if this proposal is wise. For example, if developers would only want to achieve a good effectiveness of their tests, our results highlight that the proposal would make sense if we consider that the creation of unit tests is more work than the creation of integration tests (e.g., because of the creation of mocks). If the goal is to be able to pinpoint the source of bugs (i.e., have a lower defect-locality) than the proposal would not be favorable, as unit tests are better in this case. However, we would advise developers to follow the proposal, as the savings in development time for unit tests are non-neglectable, but unit tests should still be developed for the complex parts of a software system.

8.3. Threats to Validity and Validation Procedures

In this section, we discuss the construct, external, and internal validity of our study together with the validation procedures that we have taken to measure or counter those threats.

8.3.1. Construct Validity

Construct validity threats are concerned with the degree to which our analysis measures what we intended it to measure [413]. During our research we developed tools and plug-ins for the data collection, as well as analysis scripts for our data analysis. Developing this amount of code raises the possibilities of bugs in the code that have an influence on our study. Nevertheless, we took several measures to minimize this threat. As the coverage collection is one essential part of our study (e.g., to detect the test level), we have chosen a library with a high quality and maturity. Hence, the possibility of bugs that have a big influence on the results is lowered. Furthermore, we have chosen a mutation testing framework that is mature and often used in research, e.g. in [414, 415]. It is less likely that assessment of the mutation detection capabilities is not working correctly. In addition, we manually inspected every problem that occurred during the mutation testing. We found that PIT sometimes threw an error, because a test was failing. However, this occurred only for 35 out of 36435 (i.e., 0.1%) analyzed test cases. We wrote tests for all of our tools to reduce the possibility of bugs and gives us confidence in our implementations. In addition, we used manually sampled data to test all used and created tools to determine if they are working correctly.

The used test level classification scheme might also have an influence on our results. We decided to be as close as possible to the definitions and therefore designed and implemented approaches that represent the test level classifications of the IEEE and ISTQB. While we did our experiments with these two different test level classifications, other classifications might produce different results.

For the calculation of the defect-locality of tests we integrate calls to our CallHelper (Section 4.2.10) at places where mutants were integrated before as the mutation detection capabilities were calculated. Thus, to validate the calculation of the defect-locality we can perform the following process: we know that all killed and survived mutants have been covered by the tests (by definition). Therefore, we evaluated, if all mutation results of killed/survived mutants have a defect-locality calculated for them, i.e. the test covered the place where the mutant was integrated. In the mean 1.53 % of all killed/survived mutants were not covered by the test. We determined two different reasons for this: first, PIT does not handle parameterized tests well. Hence, if the test should only be executed with one parameter, PIT executes the test for all given parameters. Therefore, if DCD executes the test with only one parameter and queries the mutation detection capabilities of this test, it gets the combined capabilities of the test executed for all parameters. Therefore, some parts might be covered with one parameter, but not with another one. We addressed this issue with PIT during our analysis by merging the results for parameterized tests (Section 6.2). Second, PIT sometimes reports the wrong line number of an integrated defect for multi-line statements, which then results in an incorrect placement of a call to the CallHelper.

Mutant redundancy is a substantial threat to the construct validity of our study. We counter this thread by calculating the disjoint mutant set, as proposed by Kintis et al. [219] and use it within our analysis. Nevertheless, this set is only an approximation and other mutants might fit better.

Another threat to the construct validity of our study is the choice of the defect classification scheme. A different classification scheme might produce different results. However, we needed a scheme that works on source code level, as we also wanted to classify integrated mutants that do not have any supplementary material, e.g., a bug report. Nevertheless, the defect classification of our mutants can be wrong for some of them. The re-integration of the defect for the classification might be problematic, if several statements are on one line. Hence, it might occur that not the correct statement gets modified and therefore the wrong classification is produced.

Our choice of the differences between unit and integration tests might not be complete. While we looked through the standard literature and collected all differences from them, there might be differences between unit and integration tests that we have overlooked or that are reported in other literature. However, we have chosen our literature based on our knowledge of the field and added often mentioned literature.

8.3.2. External Validity

External validity threats are concerned with the ability to generalize our results [413]. Parts of our study were only executed for Java projects. For the results of these parts we do not know if they are representative for other programming languages too. Although, we took a larger sample of projects than most other empirical studies [270], we cannot conclude that our results are representative for all Java projects. The problem with our approach is that

it needs a compilable release as we run the tests to gather the coverage data. Nevertheless, a compilable release of the project is often not given [215]. It is also very time-consuming as some manual work is required. Other parts of our study were executed with a second programming language, which raises the external validity of our results.

Another threat to the external validity of our study is that we only use open-source projects. We can not be sure that our results are representative for closed source projects too. Nevertheless, nine of 27 projects are mostly industry-driven, i.e. developed by a company.

8.3.3. Internal Validity

Internal validity threats are concerned with the ability to draw conclusion from the relation between causes and effects [413]. While we tried to create an isolated environment for our study (e.g., by using mutation testing) we might have overlooked some influencing variable that we did not account for. We applied different mechanisms to counter the influences that we know of, e.g., the normalization of results to calculate the defect detection capabilities (Section 6.2).

Furthermore, with our current approach we are not able to differentiate integration and system tests. It can happen that a system test gets classified as an integration test. To reduce this threat, we only included projects that are libraries or frameworks, because the possibility that they have systems tests is lower as for applications.

Another threat to the internal validity of our study is the execution of our statistical tests. We rely on the accurate implementation of the algorithms that we used during our analysis. Hence, we use a well known public library (i.e., SciPy [416]) to perform our statistical testing.

Within our qualitative analysis we analyzed different resources like research papers, developer comments, and other internet resources. However, another methodology or paper that we might have overlooked could produce different results. Nevertheless, to ease this threat, our analysis is not only based on research papers to create a “scientific view”, but also includes other resources to create an “practitioners view”.

9. Conclusion

In this section, we conclude the thesis. Therefore, we provide a summary and give an outlook on potential future work.

9.1. Summary

In this thesis, we presented a qualitative and quantitative analysis of the differences between unit and integration tests. At first, we analyzed the distribution of unit and integration tests in open-source software projects according to different definitions. Afterwards, we explored and analyzed six differences between unit and integration tests, that were mentioned in the standard literature. Three of these differences were analyzed quantitatively, the others qualitatively.

We designed and implemented several approaches to collect data from software projects. We developed an approach to classify software tests into unit and integration tests based on the definitions of the IEEE and ISTQB and the developer classification. Furthermore, we designed approaches to collect the TestLOC and pLOC of software tests. We also collected the defect detection capabilities of tests, where we applied mutation testing in combination with an approach to classify the mutants into different defect classes. Moreover, we designed an approach to extract the defect-locality of software tests using artificial defects. All of these approaches are combined into two different frameworks, which are open-source and free to use for the research community. This facilitates the replication of our study and enables other researchers to contribute to the body of knowledge of software testing research.

The quantitative analysis was done via a case study including 27 Java and Python projects with more than 49 000 tests. We classified tests into unit and integration tests according to the definitions of the IEEE and ISTQB and collected the above mentioned data from them. We found that most current open-source projects possess more integration than unit tests, if tests are classified according to the above mentioned definitions. Nevertheless, there are differences in the number of unit and integration tests between the developer classification and the classification using the definitions of the IEEE and ISTQB. Our quantitative analysis of the execution time reveled no differences between unit and integration tests for the execution time per covered line of code. More surprisingly, our results indicate that there is no difference in the overall and defect type specific effectiveness between unit and integration tests. The last part of our quantitative analysis was the assessment of the defect-locality of unit and integration tests. Within this part, we found a statistically significant difference

between unit and integration tests for the IEEE and ISTQB definitions only, showing that unit tests are more likely to pinpoint the source of a defect.

For the qualitative analysis, we reviewed related literature out of the research and industrial perspective. We created a holistic view on the analyzed differences at hand. Our qualitative analysis highlighted, that we are missing research in most of the analyzed fields to make scientifically grounded conclusions. The results of our analysis of the test execution automation showed that unit, as well as integration tests are executed automatically. There exist different approaches for the execution automation of unit and integration tests, e.g., CI systems. The results of our qualitative analysis of the test objective highlighted, that efficiency testing is mostly done on system level, while approaches exist that can be used for unit and integration level.We also identified a need for automated efficiency testing on lower test levels. Maintainability testing must be done on all levels to assess all aspects of maintainability. Nowadays projects often only make use of the MI to assess the maintainability of software, which is rather limited. More research is missing in this direction. The last aspect that we analyzed to assess the differences in the test objective is the use of robustness testing on unit and integration level. We found that robustness testing is well researched and done on unit, integration, and system level. The last difference that we analyzed qualitatively is the difference in test costs. We found that the current research on this topic cannot provide a definite answer to the question if unit or integration tests are more costly in their development, maintenance, and execution. Nevertheless, the analysis of the experience of developers highlighted, that integration tests are more costly. We are missing empirical studies with real software projects on this topic to come to a definite conclusion.

9.2. Outlook

There are several open problems and possible improvements that provide opportunities for future research. Our quantitative analysis can be improved in several ways. Overall, we could use more projects to assess if our results are generalizable. This would improve the external validity of our results and could provide further insights. Furthermore, our analysis could be done with other types of projects. Within this thesis, we focused on frameworks and libraries, but extending the focus to also include applications could give us interesting findings. Especially, as the tests of frameworks and libraries are rather different from tests for an application. Currently, we performed our analysis on open-source projects only, because we do not have industrial data available, but the use of industrial project data for the quantitative analysis could provide further insights. A comparison of the results of the quantitative analysis between open-source and industrial projects could present especially interesting results, as tests are often developed differently in an industrial context.

Future research could also include a manual analysis of test cases. This, in addition to our automated quantitative analysis, could provide insights into the reasons for our results. For example, we could manually analyze test cases and their (not) detected defects and defect

types. This could help us to understand why certain tests are not detecting certain defect types. These results could be used to develop tests that detect specific defect types.

Furthermore, we could perform our quantitative analysis on different releases of the same project. This way, we could assess how (and if) the results are changing during the evolution of a software. For example, it would be interesting to assess if there are more unit tests for a software in the beginning of the development in contrast to later versions. Moreover, checking when and if there is a transition of tests from a unit to an integration tests (or vice versa) could provide us with insights in software testing practices. The results from this analysis could help us to guide the evolution of software tests.

Another possible research direction is to assess, if the development paradigm has an influence on the distribution and quality of tests on the different test levels. For example, we could apply our approach to projects that follow the Test-Driven Development (TDD) paradigm and compare these results with results from other projects. This way we could assess the influence of TDD on the number of unit and integration tests, as well as its influence on, e.g., the effectiveness of these tests.

There are also several opportunities to improve our frameworks. Currently, our COMFORT framework is not able to differentiate integration from system tests. This could be addressed by developing a methodology to differentiate those test types from each other. One approach could be that the test is analyzed to evaluate if it is assessing the software through its main interface (e.g., the main method of a Java program). If this is the case, the test is most likely designed as a system test and as an integration test otherwise. A further improvement could be the extension of our defect classification approach. In addition to the classification of the defects in several defect classes, a severity could be assigned to each defect. This way, we could include the severity of defects into our analysis to evaluate which defects with which severity unit and/or integration tests detect.

We were able to provide a qualitative analysis of the differences between unit and integration tests regarding the test execution automation, the test objective, and the test costs. Further work can be a quantitative evaluation of these aspects. This might not be easily possible, as data might not be available for open-source projects. For example, evaluating the costs of tests is only possible using industrial data. Furthermore, we could repeat our qualitative analysis using a different methodology (e.g., conducting a Systematic Literature Review (SLR) for each difference).

In addition to the above mentioned directions and research opportunities, we plan to perform a developer study on unit and integration testing practices. We hope for feedback from developers and how they use unit and integration tests in their work. This feedback could help us to understand why developers classify their tests differently and not according to the definitions. The results of this study could lead to new definitions for unit and integration tests, that better reflect the current development reality. In connection to this study, the usage of unit and integration tests in different development models or phases would be interesting to assess. We could compare the usage of both test types in a classical and agile development environment to gather information of their usage and the differences.

Bibliography

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[2] Xi'an Jiaotong University, "ICST 2019," http://icst2019.xjtu.edu.cn, 2018, [accessed 13-November-2018].

[3] Organizing Committee ISSTA 2019, "ISSTA 2019," https://conf.researchr.org/home/issta-2019, 2018, [accessed 13-November-2018].

[4] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2004, pp. 273–281.

[5] T. Dyba, B. A. Kitchenham, and M. Jorgensen, "Evidence-based software engineering for practitioners," *IEEE Software*, vol. 22, no. 1, pp. 58–65, 2005.

[6] J. S. Molléri, K. Petersen, and E. Mendes, "Cerse-catalog for empirical research in software engineering: A systematic mapping study," *Information and Software Technology*, 2018.

[7] S. Beecham, D. Bowes, and K.-J. Stol, "Introduction to the ease 2016 special section: Evidence-based software engineering: Past, present, and future," *Information and Software Technology*, vol. 89, pp. 14 – 18, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917303877

[8] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering 2007*. IEEE Computer Society, 2007, pp. 85–103.

[9] A. Spillner, T. Linz, and H. Schaefer, *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, Inc., 2014.

[10] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[11] I. Sommerville, *Software engineering*, 9th ed. USA: Addison-Wesley Publishing Company, 2010.

[12] J. Ludewig and H. Lichter, *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013.

[13] B. Beizer, *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[14] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[15] P. Bourque, R. E. Fairley *et al.*, *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[16] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," *Empirical Software Engineering*, vol. 11, no. 1, pp. 5–31, 2006.

[17] W. Royce, *Software project management*. Pearson Education India, 1999.

[18] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, pp. 193–202.

[19] K. Taneja and T. Xie, "Diffgen: Automated regression unit-test generation," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2008, pp. 407–410.

[20] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011, pp. 364–374.

[21] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 417–420.

[22] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[23] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[24] D. Xu, W. Xu, M. Tu, N. Shen, W. Chu, and C.-H. Chang, "Automated integration testing using logical contracts," *IEEE Transactions on Reliability*, vol. 65, no. 3, pp. 1205–1222, 2016.

[25] R. Lachmann, S. Lity, M. Al-Hajjaji, F. Fürchtegott, and I. Schaefer, "Fine-grained test case prioritization for integration testing of delta-oriented software product lines," in *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. ACM, 2016, pp. 1–10.

[26] D. Holling, A. Hofbauer, A. Pretschner, and M. Gemmar, "Profiting from unit tests for integration testing," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 353–363.

[27] IEEE, "Systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, 2010.

[28] International Software Testing Qualification Board, "International Software Testing Qualification Board Glossary," http://www.astqb.org/glossary/search/unit, [accessed 13-November-2018].

[29] K. C. Dodds, "Write tests. Not too many. Mostly integration." https://blog.kentcdodds.com/write-tests-not-too-many-mostly-integration-5e8c7fff591c, 2017, [accessed 13-November-2018].

[30] J. O. Coplien, "Why most unit testing is waste," http://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf, 2014, [accessed 13-November-2018].

[31] ——, "Seque," http://rbcs-us.com/documents/Segue.pdf, 2014, [accessed 13-November-2018].

[32] E. Kiss, "Lean testing or why unit tests are worse than you think," https://blog.usejournal.com/lean-testing-or-why-unit-tests-are-worse-than-you-think-b6500139a009, 2018, [accessed 13-November-2018].

[33] M. Sustrik, "Unit test fetish," http://250bpm.com/blog:40, 2014, [accessed 13-November-2018].

[34] Oracle, "Java API: Instrumentation," https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html, [accessed 13-November-2018].

[35] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured programming*. Academic Press Ltd., 1972.

[36] B. W. Boehm, "Software engineering: R&d trends and defense needs," *Research Directions in Software Technology*, 1979.

[37] ——, "Verifying and validating software requirements and design specifications," *IEEE Software*, vol. 1, no. 1, p. 75, 1984.

[38] F. Trautsch and J. Grabowski, "Are there any unit tests? an empirical study on unit testing in open source python projects," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 207–218.

[39] G. K. Thiruvathukal, K. Läufer, and B. Gonzalez, "Unit testing considered useful," *Computing in Science and Engineering*, vol. 8, no. 6, pp. 76–87, 2006.

[40] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," *Advances in Computers*, 2017.

[41] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[42] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 354–365.

[43] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "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)*. IEEE Press, 2015, pp. 936–946.

[44] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Quantitative evaluation of mutation operators for ws-bpel compositions," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2010, pp. 142–150.

[45] A. Derezinska and K. Kowalski, "Object-oriented mutation applied in common intermediate language programs originated from c," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 342–350.

[46] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for c++ with the mucpp mutation system," *Information and Software Technology*, vol. 81, pp. 169–184, 2017.

[47] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013, pp. 74–83.

[48] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–10.

[49] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, vol. 81, pp. 154–168, 2017.

[50] R. Abraham and M. Erwig, "Mutation operators for spreadsheets," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 94–108, 2009.

[51] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans, "Towards security-aware mutation testing." in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 97–102.

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

[53] B. J. Garvin and M. B. Cohen, "Feature interaction faults revisited: An exploratory study," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2011, pp. 90–99.

[54] F. Belli, M. Beyazit, T. Takagi, and Z. Furukawa, "Mutation testing of" go-back" functions based on pushdown automata," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2011, pp. 249–258.

[55] R. Gopinath and E. Walkingshaw, "How good are your types? using mutation analysis to evaluate the effectiveness of type annotations," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2017, pp. 122–127.

[56] R. Jabbarvand and S. Malek, "μdroid: an energy-aware mutation testing framework for android," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 208–219.

[57] P. Arcaini, A. Gargantini, and E. Riccobene, "Mutrex: A mutation-based generator of fault detecting strings for regular expressions," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2017, pp. 87–96.

[58] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2010, pp. 90–99.

[59] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 435–444.

[60] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "How hard does mutation analysis have to be, anyway?" in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 216–227.

[61] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2014, pp. 11–20.

[62] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 351–360.

[63] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt, "Experimental evaluation of sdl and one-op mutation for c," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2014, pp. 203–212.

[64] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 3. ACM, 1993, pp. 139–148.

[65] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2010, pp. 121–130.

[66] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[67] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 449–452.

[68] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," *Information and Software Technology*, vol. 54, no. 9, pp. 915–932, 2012.

[69] V. H. Durelli, J. Offutt, and M. E. Delamaro, "Toward harnessing high-level language virtual machines for further speeding up weak mutation testing," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 681–690.

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

[71] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 308–333, 2018.

[72] M. Kintis and N. Malevris, "Using data flow patterns for equivalent mutant detection," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2014, pp. 196–205.

[73] ——, "Medic: A static analysis framework for equivalent mutant identification," *Information and Software Technology*, vol. 68, pp. 1–17, 2015.

[74] R. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[75] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.

[76] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011, pp. 212–222.

[77] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2013, pp. 235–245.

[78] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2012, pp. 11–20.

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

[80] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proceedings of the 18th International Symposium on Software testing and Analysis (ISSTA)*. ACM, 2009, pp. 69–80.

[81] D. Schuler and A. Zeller, "(un-) covering equivalent mutants," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2010, pp. 45–54.

[82] ——, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353–374, 2013.

[83] M. P. Usaola, P. R. Mateo, and B. P. Lamancha, "Reduction of test suites using mutation," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2012, pp. 425–438.

[84] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 46–57.

[85] C. D. Nguyen, A. Marchetto, and P. Tonella, "Change sensitivity based prioritization for audit testing of webservice compositions," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 357–365.

[86] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "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)*. IEEE Press, 2017, pp. 597–608.

[87] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *International Conference on Software Engineering (ICSE)*, 2018.

[88] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[89] ——, "Using mutants to locate" unknown" faults," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 691–700.

[90] ——, "Effective fault localization via mutation analysis: A selective mutation approach," in *Proceedings of the 29th Annual Symposium on Applied Computing*. ACM, 2014, pp. 1293–1300.

[91] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2010, pp. 65–74.

[92] ——, "Combining mutation and fault localization for automated program debugging," *Journal of Systems and Software*, vol. 90, pp. 45–60, 2014.

[93] Stat Trek, "Populations and samples," https://stattrek.com/sampling/populations-and-samples.aspx, 2018, [accessed 13-November-2018].

[94] U. Kuckartz, S. Rädiker, T. Ebert, and J. Schehl, *Statistik: eine verständliche Einführung*. Springer-Verlag, 2013.

[95] G. Kanji, *100 Statistical tests*, ser. 100 Statistical Tests. SAGE Publications, 2006. [Online]. Available: https://books.google.de/books?id=M-_WDH-06voC

[96] Stat Trek, "How to Test Hypotheses," https://stattrek.com/hypothesis-test/how-to-test-hypothesis.aspx?Tutorial=AP, 2018, [accessed 13-November-2018].

[97] H. R. Neave, "The teachrng of hypothesis-testing," *Journal of Applied Statistics*, vol. 3, no. 1, pp. 55–63, 1976.

[98] D. J. Benjamin, J. O. Berger, M. Johannesson, B. A. Nosek, E.-J. Wagenmakers, R. Berk, K. A. Bollen, B. Brembs, L. Brown, C. Camerer *et al.*, "Redefine statistical significance," *Nature Human Behaviour*, 2017.

[99] Minitab Inc., "What is a test statistic?" https://support.minitab.com/en-us/minitab-express/1/help-and-how-to/basic-statistics/inference/supporting-topics/basics/what-is-a-test-statistic, 2017, [accessed 13-November-2018].

[100] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[101] W. Daniel, *Applied nonparametric statistics*, ser. The Duxbury advanced series in statistics and decision sciences. PWS-Kent Publ., 1990. [Online]. Available: https://books.google.de/books?id=0hPvAAAAMAAJ

[102] K. P. F.R.S., "X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900. [Online]. Available: https://doi.org/10.1080/14786440009463897

[103] M. B. Brown and A. B. Forsythe, "Robust tests for the equality of variances," *Journal of the American Statistical Association*, vol. 69, no. 346, pp. 364–367, 1974.

[104] H. Levene, "Robust tests for equality of variances," *Contributions to probability and statistics. Essays in honor of Harold Hotelling*, pp. 279–292, 1961.

[105] S. F. Olejnik and J. Algina, "Type i error rates and power estimates of selected parametric and nonparametric tests of scale," *Journal of Educational Statistics*, vol. 12, no. 1, pp. 45–61, 1987.

[106] G. V. Glass and K. D. Hopkins, "Statistical methods in education and psychology," *Psyccritiques*, vol. 41, no. 12, p. 1224, 1996.

[107] Student, "The probable error of a mean," *Biometrika*, vol. 6, no. 1, pp. 1–25, 1908. [Online]. Available: http://biomet.oxfordjournals.org/content/6/1/1.short

[108] B. L. Welch, "The generalization of student's problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1/2, pp. 28–35, 1947.

[109] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, pp. 50–60, 1947.

[110] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[111] Stat Trek, "Power of a hypothesis test," https://stattrek.com/hypothesis-test/power-of-test.aspx?Tutorial=AP, 2018, [accessed 13-November-2018].

[112] J. Cohen, "Statistical power analysis for the behavioral sciences (revised ed.)," 1977.

[113] S. S. Sawilowsky, "New effect size rules of thumb," *Journal of Modern Applied Statistical Methods*, vol. 8, no. 2, pp. 597–599, 2009.

[114] R. C. Mittelhammer, G. G. Judge, and D. J. Miller, *Econometric foundations pack with CD-ROM*. Cambridge University Press, 2000, vol. 1.

[115] M. Aickin and H. Gensler, "Adjusting for multiple testing when reporting research results: the bonferroni vs holm methods." *American journal of public health*, vol. 86, no. 5, pp. 726–728, 1996.

[116] J. J. Goeman and A. Solari, "Multiple hypothesis testing in genomics," *Statistics in medicine*, vol. 33, no. 11, pp. 1946–1978, 2014.

[117] A. V. Frane, "Are per-family type i error rates relevant in social and behavioral science?" *Journal of Modern Applied Statistical Methods*, vol. 14, no. 1, p. 5, 2015.

[118] G. Orellana, G. Laghari, A. Murgia, and S. Demeyer, "On the differences between unit and integration testing in the travistorrent dataset," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 451–454.

[119] Apache Software Foundation, "Maven Surefire Plugin Website," http://maven.apache.org/surefire/maven-surefire-plugin/, 2017, [accessed 13-November-2018].

[120] ——, "Maven FailSafe Plugin Website," http://maven.apache.org/surefire/maven-failsafe-plugin/, 2017, [accessed 13-November-2018].

[121] T. Kanstrén, "Towards a deeper understanding of test coverage," *Journal of Software: Evolution and Process*, vol. 20, no. 1, pp. 59–76, 2008.

[122] C. Ullenboom, *Java 7 - Mehr als eine Insel*. Galileo Press, 2011.

[123] H. Mills, "On the statistical validation of compute programs," IBM Federal Syst. Division, Gaithersburg, MD, FSC-72-6015, Tech. Rep., 1972.

[124] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[125] B. Meek and K. Siu, "The effectiveness of error seeding," *ACM Sigplan Notices*, vol. 24, no. 6, pp. 81–89, 1989.

[126] M. J. Harrold, A. J. Offutt, and K. Tewary, "An approach to fault modeling and fault seeding using the program dependence graph," *Journal of Systems and Software*, vol. 36, no. 3, pp. 273–295, 1997.

[127] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Software: Practice and Experience*, vol. 26, no. 2, pp. 165–176, 1996.

[128] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, p. 54, 2012.

[129] Y. Jia, F. Wu, M. Harman, and J. Krinke, "Genetic improvement using higher order mutation," in *Proceedings of the Companion Publication of the Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 803–804.

[130] W. B. Langdon, B. Y. H. Lam, M. Modat, J. Petke, and M. Harman, "Genetic improvement of gpu software," *Genetic Programming and Evolvable Machines*, vol. 18, no. 1, pp. 5–44, 2017.

[131] A. J. Offutt and R. H. Untch, *Mutation 2000: Uniting the orthogonal*. Springer, 2001, pp. 34–44.

[132] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

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

[134] K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, and M. Harman, "An analysis of the relationship between conditional entropy and failed error propagation in software testing," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 573–583.

[135] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3. ACM, 1996, pp. 158–171.

[136] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 402–411.

[137] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

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

[139] A. S. Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011, pp. 342–352.

[140] J. H. Hayes, "Building a requirement fault taxonomy: Experiences from a nasa verification and validation research project," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2003, pp. 49–59.

[141] Y. Zhao, H. Leung, Y. Yang, Y. Zhou, and B. Xu, "Towards an understanding of change types in bug fixing code," *Information and Software Technology*, vol. 86, pp. 37–53, 2017.

[142] X. Xia, D. Lo, X. Wang, and B. Zhou, "Automatic defect categorization based on fault triggering conditions," in *Proceedings of the 19th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2014, pp. 39–48.

[143] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[144] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[145] J. H. Hayes, C. I. Raphael, V. K. Surisetty, and A. Andrews, "Fault links: exploring the relationship between module and fault types," in *European Dependable Computing Conference*. Springer, 2005, pp. 415–434.

[146] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Proceedings of the International Conference on Software Maintenance.* IEEE, 2000, pp. 131–142.

[147] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[148] R. Abreu, A. González, P. Zoeteweij, and A. J. van Gemund, "Automatic software fault localization using generic program invariants," in *Proceedings of the ACM Symposium on Applied computing*. ACM, 2008, pp. 712–717.

[149] T. Wang and A. Roychoudhury, "Automated path generation for software fault localization," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2005, pp. 347–351.

[150] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 296–305.

[151] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 433–436.

[152] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION)*. IEEE, 2007, pp. 89–98.

[153] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[154] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 35–44.

[155] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, pp. 383–392.

[156] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *European Conference on Object-Oriented Programming*. Springer, 2005, pp. 528–550.

[157] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24rd International Conference on Software Engineering (ICSE)*. IEEE, 2002, pp. 467–477.

[158] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *International Conference on Quality Software (QSIC)*. IEEE, 2013, pp. 103–112.

[159] P. S. Kochhar, F. Thung, D. Lo, and J. Lawall, "An empirical study on the adequacy of testing in open source projects," in *Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1. IEEE, 2014, pp. 215–222.

[160] D. Ma'ayan, "The quality of junit tests: an empirical study report," in *International Workshop on Software Qualities and their Dependencies (SQUADE)*. IEEE, 2018, pp. 33–36.

[161] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2008, pp. 220–229.

[162] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[163] M. Wacker, "Just say no to more end-to-end tests," https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html, 2015, [accessed 13-November-2018].

[164] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2009, pp. 209–218.

[165] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, "Ezunit: A framework for associating failed unit tests with potential programming errors," in *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2007, pp. 101–104.

[166] A. Van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.

[167] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2005, pp. 10–pp.

[168] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017.

[169] Apache Software Foundation, "Apache Software Foundation Homepage," https://www.apache.org/, 2018, [accessed 13-November-2018].

[170] TIOBE software BV, "TIOBE Index for August 2018," https://www.tiobe.com/tiobe-index/, 2018, [accessed 13-November-2018].

[171] Apache Software Foundation, "Maven Project Homepage," https://maven.apache.org/, 2017, [accessed 13-November-2018].

[172] JUnit Team, "JUnit Homepage," http://junit.org/junit5/, 2017, [accessed 13-November-2018].

[173] C. Beust, "TestNG Documentation," http://testng.org/doc/, 2015, [accessed 13-November-2018].

[174] Python Software Foundation, "Unittest library," https://docs.python.org/3/library/unittest.html, [accessed 13-November-2018].

[175] H. Krekel, "Pytest Documentation," https://docs.pytest.org/en/latest/, 2015, [accessed 13-November-2018].

[176] Google LLC, "Fundamentals of testing (Android)," https://developer.android.com/training/testing/fundamentals, 2018, [accessed 13-November-2018].

[177] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 334–344.

[178] ——, "Improved list of popular github repositories," https://doi.org/10.5281/zenodo.804473, 2017, [accessed 13-November-2018].

[179] MVN Repository, "MVN Repository - Most Popular," https://mvnrepository.com/popular, 2018, [accessed 13-November-2018].

[180] GitHub Inc., "GitHub Homepage," https://github.com/, 2018, [accessed 13-November-2018].

[181] T. Preston-Werner, "Semantic Versioning 2.0.0," http://semver.org/, 2017, [accessed 13-November-2018].

[182] Apache Software Foundation, "commons-beanutils GitHub," https://github.com/apache/commons-beanutils, 2018, [accessed 13-November-2018].

[183] ——, "commons-codec GitHub," https://github.com/apache/commons-codec, 2018, [accessed 13-November-2018].

[184] ——, "commons-collections GitHub," https://github.com/apache/commons-collections, 2018, [accessed 13-November-2018].

[185] ——, "commons-io GitHub," https://github.com/apache/commons-io, 2018, [accessed 13-November-2018].

[186] ——, "commons-lang GitHub," https://github.com/apache/commons-lang, 2018, [accessed 13-November-2018].

[187] ——, "commons-math GitHub," https://github.com/apache/commons-math, 2018, [accessed 13-November-2018].

[188] Alibaba, "druid GitHub," https://github.com/alibaba/druid, 2018, [accessed 13-November-2018].

[189] ——, "fastjson GitHub," https://github.com/alibaba/fastjson, 2018, [accessed 13-November-2018].

[190] Google LLC, "gson GitHub," https://github.com/google/gson, 2018, [accessed 13-November-2018].

[191] ——, "guice GitHub," https://github.com/google/guice, 2018, [accessed 13-November-2018].

[192] B. Wooldridge, "HikariCP GitHub," https://github.com/brettwooldridge/HikariCP, 2018, [accessed 13-November-2018].

[193] FasterXML LLC, "jackson-core GitHub," https://github.com/FasterXML/jackson-core, 2018, [accessed 13-November-2018].

[194] D. Gilbert, "jfreechart GitHub," https://github.com/jfree/jfreechart, 2018, [accessed 13-November-2018].

[195] Joda.org, "joda-time GitHub," https://github.com/JodaOrg/joda-time, 2018, [accessed 13-November-2018].

[196] J. Hedley, "jsoup GitHub," https://github.com/jhy/jsoup, 2018, [accessed 13-November-2018].

[197] MyBatis, "mybatis-3 GitHub," https://github.com/mybatis/mybatis-3, 2018, [accessed 13-November-2018].

[198] Google LLC, "zxing GitHub," https://github.com/google/zxing, 2018, [accessed 13-November-2018].

[199] G. Cox, "ChatterBot GitHub," https://github.com/gunthercox/ChatterBot, 2018, [accessed 13-November-2018].

[200] J. McKinney and N. Bedi, "csvkit GitHub," https://github.com/wireservice/csvkit, 2018, [accessed 13-November-2018].

[201] Douban Inc., "dpark GitHub," https://github.com/douban/dpark, 2018, [accessed 13-November-2018].

[202] Yelp.com, "mrjob GitHub," https://github.com/Yelp/mrjob, 2018, [accessed 13-November-2018].

[203] Networkx Community, "networkx GitHub," https://github.com/networkx/networkx, 2018, [accessed 13-November-2018].

[204] Pylons Project, "pyramid GitHub," https://github.com/Pylons/pyramid, 2018, [accessed 13-November-2018].

[205] Python-telegram-bot Community, "python-telegram-bot GitHub," https://github.com/python-telegram-bot/python-telegram-bot, 2018, [accessed 13-November-2018].

[206] J. Leidel and S. Colvin, "rq GitHub," https://github.com/rq/rq, 2018, [accessed 13-November-2018].

[207] K. Tuure, J. Dnns, and R. Kracekumar, "schematics GitHub," https://github.com/schematics/schematics, 2018, [accessed 13-November-2018].

[208] Scrapy Community, "scrapy GitHub," https://github.com/scrapy/scrapy, 2018, [accessed 13-November-2018].

[209] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 322–333.

[210] J. Janák, "Issue tracking systems," Ph.D. dissertation, Masarykova univerzita, Fakulta informatiky, 2009.

[211] Apache Software Foundation, "Bugzilla Homepage," https://www.bugzilla.org/, [accessed 13-November-2018].

[212] Atlassian, "Jira Software," https://de.atlassian.com/software/jira, 2018, [accessed 13-November-2018].

[213] Oracle, "What is a package?" https://docs.oracle.com/javase/tutorial/java/concepts/package.html, 2017, [accessed 13-November-2018].

[214] Python Software Foundation, "Python Documentation: Packages," https://docs.python.org/3/tutorial/modules.html#packages, 2018, [accessed 13-November-2018].

[215] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, 2017.

[216] S. Faber, "Mockito Mocking Framework," https://site.mockito.org/, 2018, [accessed 13-November-2018].

[217] H. Tremblay, "Easymock mocking framework," http://easymock.org/, 2018, [accessed 13-November-2018].

[218] Nose Developers, "Nosetest Documentation," http://nose.readthedocs.io/en/latest/, 2016, [accessed 13-November-2018].

[219] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, pp. 1–38, 2017.

[220] Sixty North AS, "Cosmic-ray documentation," http://cosmic-ray.readthedocs.io/en/latest/index.html, 2001, [accessed 13-November-2018].

[221] I. Moore, "Jester and Pester," http://jester.sourceforge.net/, 2001, [accessed 13-November-2018].

[222] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: a mutation system for java," in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 827–830.

[223] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "An experimental mutation system for java," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, pp. 1–4, 2004.

[224] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[225] n.A., "Jumble Testing Tool for Java," http://jumble.sourceforge.net/, 2001, [accessed 13-November-2018].

[226] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for java," in *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, pp. 297–298.

[227] T. Knauth, C. Fetzer, and P. Felber, "Assertion-driven development: Assessing the quality of contracts using meta-mutations," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2009, pp. 182–191.

[228] H. Madeira *et al.*, "Definition of software fault emulation operators: A field data study," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2003, p. 105.

[229] M. Gligoric, V. Jagannath, and D. Marinov, "Mutmut: Efficient exploration for mutation testing of multithreaded code," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2010, pp. 55–64.

[230] L. Madeyski and N. Radyk, "Judy–a mutation testing tool for java," *IET software*, vol. 4, no. 1, pp. 32–42, 2010.

[231] P. R. Mateo, M. P. Usaola, and J. Offutt, "Mutation at system and functional levels," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2010, pp. 110–119.

[232] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 433–436.

[233] P. Madiraju and A. S. Namin, "Paraμ–a partial and higher-order mutation tool with concurrency operators," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 351–356.

[234] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using conditional mutation to increase the efficiency of mutation analysis," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 50–56.

[235] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2013, pp. 224–234.

[236] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent java (j2se 5.0)," in *Workshop on Mutation Analysis (MUTATION)*. IEEE, 2006, p. 11.

[237] E. Omar, S. Ghosh, and D. Whitley, "Homaj: A tool for higher order mutation testing in aspectj and java," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2014, pp. 165–170.

[238] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. Le Traon, and A. Ventresque, "Assessing and improving the mutation testing practice of pit," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 430–435.

[239] A. Parsai, A. Murgia, and S. Demeyer, "Littledarwin: a feature-rich and extensible mutation testing framework for large and complex java systems," in *International Conference on Fundamentals of Software Engineering (FSEN)*. Springer, 2017, pp. 148–163.

[240] A. Derezińska and K. Hałas, "Analysis of mutation operators for the python language," in *Proceedings of the 9th International Conference on Dependability and Complex Systems (DepCoS)*. Springer, 2014, pp. 155–164.

[241] Sixty North AS, "Cosmic-ray github," https://github.com/sixty-north/cosmic-ray, 2017, [accessed 13-November-2018].

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

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

[244] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[245] ——, "Better predicate testing," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 57–63.

[246] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 720–725.

[247] P. Ammann, M. E. Delamaro, J. Offutt *et al.*, "Establishing theoretical minimal sets of mutants," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2014.

[248] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2014, pp. 176–185.

[249] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–10.

[250] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 2010, pp. 300–309.

[251] P. Anbalagan and T. Xie, "Automated generation of pointcut mutants for testing pointcuts in aspectj programs," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2008, pp. 239–248.

[252] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Mutation operators for ws-bpel 2.0," in *International Conference on Software & Systems Engineering and their Applications*, 2008.

[253] J. Boubeta-Puig, I. Medina-Bulo, and A. García-Domínguez, "Analogies and differences between mutation operators for ws-bpel 2.0 and other languages," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 398–407.

[254] J. Hu, N. Li, and J. Offutt, "An analysis of oo mutation operators," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 334–341.

[255] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2008, pp. 52–61.

[256] L. Bottaci, "Type sensitive application of mutation operators for dynamically typed programs," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2010, pp. 126–131.

[257] A. Alberto, A. Cavalcanti, M.-C. Gaudel, and A. Simão, "Formal mutation testing for circus," *Information and Software Technology*, vol. 81, pp. 131–153, 2017.

[258] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2010, pp. 132–141.

[259] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu, "An experimental evaluation of web mutation operators," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2016, pp. 102–111.

[260] J. Nanavati, F. Wu, M. Harman, Y. Jia, and J. Krinke, "Mutation testing of memory-related operators," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–10.

[261] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke, "Memory mutation testing," *Information and Software Technology*, vol. 81, pp. 97–111, 2017.

[262] M. E. Delamaro, J. Maidonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[263] M. Grechanik and G. Devanla, "Mutation integration testing," in *International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2016, pp. 353–364.

[264] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Engineering and Verification: International Summer Schools, LASER*. Springer, 2012, pp. 1–59.

[265] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011, pp. 265–275.

[266] M. E. Delamaro, J. Offutt *et al.*, "Assessing the influence of multiple test case selection on mutation experiments," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Computer Society, 2014.

[267] H. Coles, "PIT Project Homepage," http://pitest.org/, 2017, [accessed 13-November-2018].

[268] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *International Conference on Program Comprehension (ICPC*. IEEE, 2006, pp. 35–45.

[269] D. A. Wheeler, "Why open source software/free software (oss/fs, floss, or foss)? look at the numbers," 2007.

[270] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski, "Addressing problems with replicability and validity of repository mining studies through a smart data platform," *Empirical Software Engineering*, vol. 23, no. 2, pp. 1036–1083, 2018.

[271] R. Cattell, "Scalable sql and nosql data stores," *Acm Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[272] MongoDB, Inc., "MongoDB: Who Uses MongoDB," https://www.mongodb.com/who-uses-mongodb, [accessed 13-November-2018].

[273] ——, "MongoDB Documentation: Sharding," https://docs.mongodb.com/manual/sharding, [accessed 13-November-2018].

[274] ——, "MongoDB Documentation: Libraries and Drivers," https://docs.mongodb.com/manual/applications/drivers, [accessed 13-November-2018].

[275] ——, "MongoDB Documentation: Replication," https://docs.mongodb.com/manual/replication, [accessed 13-November-2018].

[276] libgit2 Community, "libgit2 Homepage," https://libgit2.github.com/, [accessed 13-November-2018].

[277] Python Software Foundation, "Python Documentation - Multiprocessing," https://docs.python.org/3.5/library/multiprocessing.html?highlight=process, [accessed 13-November-2018].

[278] F. Trautsch, "COMFORT-core GitHub," https://github.com/comfort-framework/comfort, 2018, [accessed 13-November-2018].

[279] Mountainminds GmbH & Co. KG and Contributors, "Jacoco Project Homepage," http://www.eclemma.org/jacoco/, 2017, [accessed 13-November-2018].

[280] ——, "JaCoCo FAQ," http://www.jacoco.org/jacoco/trunk/doc/faq.html, 2014, [accessed 13-November-2018].

[281] ——, "Jacoco Maven Build," https://www.jacoco.org/jacoco/trunk/doc/maven.html, 2017, [accessed 13-November-2018].

[282] ——, "Jacoco API," https://www.jacoco.org/jacoco/trunk/doc/api/index.html, 2017, [accessed 13-November-2018].

[283] ——, "Jacoco Documentation," https://www.jacoco.org/jacoco/trunk/doc/index.html, 2017, [accessed 13-November-2018].

[284] F. Trautsch, "COMFORT-jacoco-listener GitHub," https://github.com/comfort-framework/comfort-jacoco-listener, 2018, [accessed 13-November-2018].

[285] Apache Software Foundation, "Apache Maven Documentation: Using JUnit," https://maven.apache.org/surefire/maven-surefire-plugin/examples/junit.html, 2018, [accessed 13-November-2018].

[286] Mkyong.com, "Mkyon.com: JUnit - Basic annotation examples," https://www.mkyong.com/unittest/junit-4-tutorial-1-basic-usage/, 2009, [accessed 13-November-2018].

[287] Ned Batchelder, "Coverage.py project homepage," https://coverage.readthedocs.io/en/coverage-4.4.1/, 2017, [accessed 13-November-2018].

[288] F. Trautsch, "COMFORT-smother GitHub," https://github.com/comfort-framework/comfort-smother, 2018, [accessed 13-November-2018].

[289] Mountainminds GmbH & Co. KG and Contributors, "Jacoco: Java Agent," https://www.eclemma.org/jacoco/trunk/doc/agent.html, [accessed 13-November-2018].

[290] ——, "Jacoco: Implementation Design," https://www.eclemma.org/jacoco/trunk/doc/implementation.html, [accessed 13-November-2018].

[291] D. A. Wheeler, "SLOCCount," https://www.dwheeler.com/sloccount, [accessed 13-November-2018].

[292] M. Fewster and D. Graham, *Software test automation: Effective use of test execution tools*, ser. ACM Press Series. Addison-Wesley, 1999. [Online]. Available: https://books.google.de/books?id=z2QABZKTihQC

[293] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 419–423.

[294] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[295] A. S. Kumar and S. Vasavi, "Effective unit testing framework for automation of windows applications," in *International Conference on Advances in Computing (ICACCI)*. Springer, 2013, pp. 813–822.

[296] Z. Zhang, J. Thangarajah, and L. Padgham, "Automated unit testing for agent systems." *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, vol. 7, pp. 10–18, 2007.

[297] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim, "Automated unit testing of large industrial embedded software using concolic testing," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 519–528.

[298] S. A. Jolly, V. Garousi, and M. M. Eskandar, "Automated unit testing of a scada control software: an industrial case study based on action research," in *International*

Conference on Software Testing, Verification and Validation (ICST). IEEE, 2012, pp. 400–409.

[299] L. Williams, G. Kudrjavets, and N. Nagappan, "On the effectiveness of unit test automation at microsoft," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2009, pp. 81–89.

[300] S. H. Edwards, "A framework for practical, automated black-box testing of component-based software," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 97–111, 2001.

[301] E. Collins, A. Dias-Neto, and V. F. de Lucena Jr, "Strategies for agile software testing automation: An industrial experience," in *Annual Computer Software and Applications Conference Workshops (COMPSACW)*. IEEE, 2012, pp. 440–445.

[302] M. Stephens and D. Rosenberg, "Automated integration testing," in *Design Driven Testing*. Springer, 2010, pp. 253–276.

[303] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 571–579.

[304] D. Graham and M. Fewster, *Experiences of test automation: case studies of software test automation*. Addison-Wesley Professional, 2012.

[305] J. Miller, "Succeeding with automated integration tests," https://jeremydmiller.com/2015/06/25/succeeding_with_integration_testing, 2015, [accessed 13-November-2018].

[306] P. Niederwieser, "Spock Framework," http://spockframework.org/, 2018, [accessed 13-November-2018].

[307] Vector Software, Inc., "Vector Framework," https://www.vectorcast.com/automated-integration-testing, 2018, [accessed 13-November-2018].

[308] ConSol Software GmbH, "Citrus Framework," https://citrusframework.org/, 2018, [accessed 13-November-2018].

[309] LDRA, "LDRA Framework," https://ldra.com/, 2018, [accessed 13-November-2018].

[310] S. Vaaraniemi, "The benefits of automated unit testing," https://www.codeproject.com/Articles/5404/The-benefits-of-automated-unit-testing, 2003, [accessed 13-November-2018].

[311] E. Dietrich and R. Williams, "Unit testing and test automation: Two things you're Not doing enough of," https://dzone.com/articles/unit-testing-and-test-automation-two-things-youre, 2018, [accessed 13-November-2018].

[312] JetBrains, "Configuring testing libraries," https://www.jetbrains.com/help/idea/configuring-testing-libraries.html, 2018, [accessed 13-November-2018].

[313] Eclipse Foundation, Inc., "Writing and running JUnit tests," https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2FgettingStarted%2Fqs-junit.htm, 2018, [accessed 13-November-2018].

[314] Atlassian, "The different types of software testing," https://www.atlassian.com/continuous-delivery/different-types-of-software-testing, 2018, [accessed 13-November-2018].

[315] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, 2000.

[316] J. Zhang and S. C. Cheung, "Automated test case generation for the stress testing of multimedia systems," *Software: Practice and Experience*, vol. 32, no. 15, pp. 1411–1435, 2002.

[317] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker, "Software performance testing based on workload characterization," in *Proceedings of the 3rd International Workshop on Software and Performance*. ACM, 2002, pp. 17–24.

[318] G. Denaro, A. Polini, and W. Emmerich, "Early performance testing of distributed software applications," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 1. ACM, 2004, pp. 94–103.

[319] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Q. Wang, J. Park, and C. Pu, "Expertus: A generator approach to automate performance testing in iaas clouds," in *International Conference on Cloud Computing (CLOUD)*. IEEE, 2012, pp. 115–122.

[320] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 1012–1021.

[321] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2009, pp. 125–134.

[322] A. Avritzer and E. Weyuker, "The automatic generation of load test suites and the assessment of the resulting software," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 705–716, 1995.

[323] A. Avritzer and E. J. Weyuker, "Deriving workloads for performance testing," *Software: Practice and Experience*, vol. 26, no. 6, pp. 613–633, 1996.

[324] V. Garousi, L. C. Briand, and Y. Labiche, "Traffic-aware stress testing of distributed systems based on uml models," in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 391–400.

[325] Riverbed Technology, "Application performance testing: Exceed user expectations," https://www.riverbed.com/de/services/services-catalog/application-performance-testing.html, 2018, [accessed 13-November-2018].

[326] Testbirds, "Load and performance testing," https://www.testbirds.com/services/quality-assurance/load-and-performance-testing, 2018, [accessed 13-November-2018].

[327] Infosys Limited, "Enhance your systems' performance for agile delivery and market leading quality," https://www.infosys.com/IT-services/validation-solutions/service-offerings/Pages/performance-testing-engineering.aspx, 2018, [accessed 13-November-2018].

[328] Cigniti, "Performance test engineering services," https://www.cigniti.com/performance-testing, 2018, [accessed 13-November-2018].

[329] Testronic Laboratories, "Performance testing," https://www.testroniclabs.com/performance-testing, 2018, [accessed 13-November-2018].

[330] RadView Software Ltd., "WebLOAD free edition," https://www.radview.com/webload-download/?utm_campaign=top-15-tools&utm_medium=top-15-tools&utm_source=softwaretestinghelp, 2018, [accessed 13-November-2018].

[331] Apache Software Foundation, "Apache JMeter," http://jmeter.apache.org, 2018, [accessed 13-November-2018].

[332] Dotcom-Monitor, Inc., "Loadview on demand performance testing," https://www.loadview-testing.com/, 2018, [accessed 13-November-2018].

[333] IBM, "Ibm rational performance tester," https://www.ibm.com/developerworks/downloads/r/rpt, 2018, [accessed 13-November-2018].

[334] Software Testing Help, "Top 15 performance testing tools of 2018: Load testing tools List," https://www.softwaretestinghelp.com/performance-testing-tools-load-testing-tools, 2018, [accessed 13-November-2018].

[335] JVM Monitor project, "JVM Monitor: Java profiler integrated with Eclipse," http://www.jvmmonitor.org, 2012, [accessed 13-November-2018].

[336] J. Sedlacek and T. Hurka, "VisualVM All-in-One Java Troubleshooting Tool," https://visualvm.github.io, 2017, [accessed 13-November-2018].

[337] Microsoft, "Performance testing guidance for web applications," https://docs.microsoft.com/en-us/previous-versions/msp-n-p/bb924375(v%3dpandp.10), 2010, [accessed 13-November-2018].

[338] M. Honda, "The art of software performance testing," https://devops.com/the-art-of-software-performance-testing, 2018, [accessed 13-November-2018].

[339] S. Evans, "Don't do performance testing in production environments only," https://techbeacon.com/dont-do-performance-testing-production-environments-only, 2018, [accessed 13-November-2018].

[340] S. Palamarchuk, "Why performance testing is necessary," https://abstracta.us/blog/performance-testing/why-performance-testing-is-necessary, 2016, [accessed 13-November-2018].

[341] Stackify, "The ultimate guide to performance testing and software testing: Testing types, performance testing Steps, best practices, and more," https://stackify.com/ultimate-guide-performance-testing-and-software-testing, 2017, [accessed 13-November-2018].

[342] ISO, "Iec 25000 software and system engineering–software product quality requirements and evaluation (square)," *International Organization for Standarization*, 2005.

[343] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proceedings of the Conference on Software Maintenance*. IEEE, 1992, pp. 337–344.

[344] F. Zhuo, B. Lowther, P. Oman, and J. Hagemeister, "Constructing and testing software maintainability assessment models," in *Proceedings of the 1st International Software Metrics Symposium*. IEEE, 1993, pp. 61–70.

[345] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[346] D. Ash, J. Alderete, P. W. Oman, and B. Lowther, "Using software maintainability models to track code health." in *Proceedings of the International Conference on Software Maintenance*, vol. 94, 1994, pp. 154–160.

[347] D. Coleman, B. Lowther, and P. Oman, "The application of software maintainability models in industrial software systems," *Journal of Systems and Software*, vol. 29, no. 1, pp. 3–16, 1995.

[348] K. D. Welker, "The software maintainability index revisited," *CrossTalk*, vol. 14, pp. 18–21, 2001.

[349] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE Computer Society, 2009, pp. 367–377.

[350] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2007, pp. 30–39.

[351] M. Broy, F. Deissenboeck, and M. Pizka, "Demystifying maintainability," in *Proceedings of the International Workshop on Software Quality*. ACM, 2006, pp. 21–26.

[352] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Proceedings of the Annual Reliability and Maintainability Symposium*. IEEE, 2002, pp. 235–241.

[353] Z. Naboulsi, "Code Metrics â€“ Maintainability Index," https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/, 2011, [accessed 13-November-2018].

[354] FrontEndART Ltd., "SourceMeter Documentation," https://www.sourcemeter.com/resources/java/, 2016, [accessed 13-November-2018].

[355] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[356] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 931–940.

[357] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?" *Information and Software Technology*, vol. 74, pp. 204–218, 2016.

[358] A. L. Jacob and S. Pillai, "Statistical process control to improve coding and code review," *IEEE Software*, no. 3, pp. 50–55, 2003.

[359] B. Vanderwal, "The five habits of maintainable unit tests," https://spin.atomicobject.com/2018/02/26/maintainable-unit-tests/, 2018, [accessed 13-November-2018].

[360] S. Resca, "Concepts of maintainable unit tests," https://samueleresca.net/2017/08/maintainable-unit-tests/, 2017, [accessed 13-November-2018].

[361] N. Kukreja, "Measuring software maintainability," https://quandarypeak.com/2015/02/measuring-software-maintainability, 2015, [accessed 13-November-2018].

[362] SonarSource S.A, "SonarQube," https://www.sonarqube.org, 2018, [accessed 13-November-2018].

[363] Checkstyle Community, "Checkstyle," http://checkstyle.sourceforge.net, 2018, [accessed 13-November-2018].

[364] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 712–721.

[365] Google Inc., "Gerrit," https://www.gerritcodereview.com, 2018, [accessed 13-November-2018].

[366] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using fiat," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, 1990.

[367] T. K. Tsai and R. K. Iyer, "Measuring fault tolerance with the ftape fault injection tool," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 1995, pp. 26–40.

[368] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A tool for the validation of system dependability properties," in *FTCS*, 1992, pp. 336–344.

[369] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[370] Z. Micskei, H. Madeira, A. Avritzer, I. Majzik, M. Vieira, and N. Antunes, "Robustness testing techniques and tools," in *Resilience Assessment and Evaluation of Computing Systems*. Springer, 2012, pp. 323–339.

[371] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," Technical report, Tech. Rep., 1995.

[372] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing," in *Proceedings of the 1st International Workshop on Random Testing*. ACM, 2006, pp. 46–54.

[373] A. K. Ghosh, M. Schmid, and V. Shah, "Testing the robustness of windows nt software," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 1998, pp. 231–235.

[374] M. Mendonca and N. Neves, "Robustness testing of the windows ddk," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2007, pp. 554–564.

[375] P. Koopman and J. DeVale, "The exception handling effectiveness of posix operating systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 837–848, 2000.

[376] P. Koopman, K. DeVale, and J. DeVale, "Interface robustness testing: Experience and lessons learned from the ballista project," *Dependability Benchmarking for Computer Systems*, vol. 72, p. 201, 2008.

[377] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[378] J.-C. Fernandez, L. Mounier, and C. Pachon, "A model-based approach for robustness testing," in *IFIP International Conference on Testing of Communicating Systems*. Springer, 2005, pp. 333–348.

[379] J. Oláh and I. Majzik, "A model based framework for specifying and executing fault injection experiments," in *International Conference on Dependability of Computer Systems*. IEEE Computer Society Press, June 2009, pp. 107–114.

[380] A. M. Memon, "An event-flow model of gui-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[381] Z. Micskei, I. Majzik, and F. Tam, "Comparing robustness of ais-based middleware implementations," in *International Service Availability Symposium*. Springer, 2007, pp. 20–30.

[382] R. Maia, L. Henriques, R. Barbosa, D. Costa, and H. Madeira, "Xception fault injection and robustness testing framework: a case-study of testing rtems," in *VI Test and Fault Tolerance Workshop*, 2005.

[383] R. Moraes, J. Duraes, R. Barbosa, E. Martins, and H. Madeira, "Experimental risk assessment and comparison using software fault injection," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2007, pp. 512–521.

[384] J. Arlat, J.-C. Fabre, and M. Rodríguez, "Dependability of cots microkernel-based systems," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 138–163, 2002.

[385] J. Fonseca, M. Vieira, and H. Madeira, "Online detection of malicious data access using dbms auditing," in *Proceedings of the ACM Symposium on Applied computing*. ACM, 2008, pp. 1013–1020.

[386] M. Vieira and H. Madeira, "A dependability benchmark for oltp application environments," in *Proceedings of the 29th International Conference on Very Large Databases*. VLDB Endowment, 2003, pp. 742–753.

[387] N. Looker, M. Munro, and J. Xu, "Ws-fit: A tool for dependability analysis of web services," in *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2004, pp. 120–123.

[388] E. Martin, S. Basu, and T. Xie, "Websob: A tool for robustness testing of web services," in *Companion to the Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007, pp. 65–66.

[389] W.-T. Tsai, X. Wei, Y. Chen, and R. Paul, "A robust testing framework for verifying web services by completeness and consistency analysis," in *International Workshop Service-Oriented System Engineering (SOSE)*. IEEE, 2005, pp. 151–158.

[390] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 12th International Conference on World Wide Web*. ACM, 2003, pp. 148–159.

[391] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *Proceedings of the 15th International Conference on World Wide Web*. ACM, 2006, pp. 247–256.

[392] S. McAllister, E. Kirda, and C. Kruegel, "Leveraging user interactions for in-depth testing of web applications," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008, pp. 191–210.

[393] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective detection of sql/xpath injection vulnerabilities in web services," in *International Conference on Services Computing (SCC)*. IEEE, 2009, pp. 260–267.

[394] N. Laranjeiro, M. Vieira, and H. Madeira, "Protecting database centric web services against sql/xpath injection attacks," in *International Conference on Database and Expert Systems Applications*. Springer, 2009, pp. 271–278.

[395] ProfessionalQA.com, "Robustness testing," http://www.professionalqa.com/robustness-testing, 2016, [accessed 13-November-2018].

[396] V. Sinclair, "Software robustness tutorial: Testing of complex telecommunications solutions," http://softtest.ie/wp-content/uploads/2017/10/Software-Robustness-Testing-of-Complex-Software-Solutions-Vincent-Sinclair.pdf, 2017, [accessed 13-November-2018].

[397] J. Bridgewater and P. Leisy, "Improving software robustness using pseudorandom test generation," https://labs.vmware.com/vmtj/improving-software-robustness-using-pseudorandom-test-generation, 2013, [accessed 13-November-2018].

[398] QA Systems GmbH, "Cantata: Flexible testing techniques," https://www.qa-systems.com/tools/cantata/flexible-testing-techniques, 2018, [accessed 13-November-2018].

[399] G. Bahmutov, " Robustness testing using proxies ," https://glebbahmutov.com/blog/robustness-testing-using-proxies, 2014, [accessed 13-November-2018].

[400] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering 2007*. IEEE Computer Society, 2007, pp. 171–187.

[401] Software Testing Help, "How to perform manual performance testing?" https://www.softwaretestinghelp.com/manual-performance-testing, 2018, [accessed 13-November-2018].

[402] B. W. Boehm *et al.*, *Software engineering economics*. Prentice-hall Englewood Cliffs (NJ), 1981, vol. 197.

[403] L. M. Karg, M. Grottke, and A. Beckhaus, "A systematic literature review of software quality cost research," *Journal of Systems and Software*, vol. 84, no. 3, pp. 415–427, 2011.

[404] G. J. Myers, "A controlled experiment in program testing and code walkthroughs/inspections," *Communications of the ACM*, vol. 21, no. 9, pp. 760–768, 1978.

[405] V. R. Basili and R. W. Selby Jr, "Comparing the effectiveness of software testing strategies," *Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili*, vol. 1, p. 300, 2005.

[406] O. Laitenberger, "Studying the effects of code inspection and structural testing on software quality," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 1998, p. 237.

[407] D. Delgado and A. Martinez, "Cost effectiveness of unit testing: A case study in a financial institution," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2013, pp. 340–347.

[408] S. King, J. Hammond, R. Chapman, and A. Pryor, "Is proof more cost-effective than testing?" *IEEE Transactions on software Engineering*, vol. 26, no. 8, pp. 675–686, 2000.

[409] J. Swett, "Why "Write Tests. Not Too Many. Mostly Integration" is bad advice," https://www.codewithjason.com/write-tests-not-too-many/, 2018, [accessed 13-November-2018].

[410] J. B. Rainsberger, "Part 2: Some Hidden Costs of Integrated Tests," https://blog.thecodewhisperer.com/permalink/part-2-some-hidden-costs-of-integrated-tests, 2009, [accessed 13-November-2018].

[411] ——, "Not Just Slow: Integration Tests are a Vortex of Doom," https://blog.thecodewhisperer.com/permalink/not-just-slow-integration-tests-are-a-vortex-of-doom, 2010, [accessed 13-November-2018].

[412] K. Kudryashov, "Economy of tests," http://stakeholderwhisperer.com/posts/2015/1/economy-of-tests, 2015, [accessed 13-November-2018].

[413] M. Gibbert, W. Ruigrok, and B. Wicki, "What passes as a rigorous case study?" *Strategic Management Journal*, vol. 29, no. 13, pp. 1465–1474, 2008.

[414] I. Saleh and K. Nagi, "Hadoopmutator: A cloud-based mutation testing framework," in *International Conference on Software Reuse*. Springer, 2015, pp. 172–187.

[415] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 342–353.

[416] SciPy developers, "SciPy Website," https://www.scipy.org/, 2017, [accessed 13-November-2018].

[417] Travis CI GmbH, "Travis CI Homepage," https://travis-ci.org/, 2017, [accessed 13-November-2018].

A. Defect Class Mappings

In Table A.1 the mappings of the CTs by Fluri et al. [268] that can be directly mapped to the defect classes by Zhao et al. [141] are depicted. This mapping is used within our case study to map the output of ChangeDistiller [268], which is applied to our generated mutants, to the defect classes by Zhao et al. [141].

Table A.2 depicts different conditions that must be met by the output of ChangeDistiller [268], i.e. the CT, CE, and PE must be of certain types, so that they are assigned a specific defect class. As the direct mapping shown in Table A.1, this information is needed to classify our generated mutants into different defect classes.

Change Type (CT)	**Defect Class**
ADDING_ATTRIBUTE_MODIFIABILITY	Data
ADDING_CLASS_DERIVABILITY	Interface
ADDING_METHOD_OVERRIDABILITY	Interface
ADDITIONAL_CLASS	Interface
ADDITIONAL_OBJECT_STATE	Data
ALTERNATIVE_PART_DELETE	Logic/Control
ALTERNATIVE_PART_INSERT	Logic/Control
ATTRIBUTE_RENAMING	Data
ATTRIBUTE_TYPE_CHANGE	Data
CLASS_RENAMING	Interface
COMMENT_DELETE	Other
COMMENT_INSERT	Other
COMMENT_MOVE	Other
COMMENT_UPDATE	Other
CONDITION_EXPRESSION_CHANGE	Logic/Control
DECREASING_ACCESSIBILITY_CHANGE	Interface
DOC_DELETE	Other
DOC_INSERT	Other
DOC_UPDATE	Other
INCREASING_ACCESSIBILITY_CHANGE	Interface
METHOD_RENAMING	Interface
PARAMETER_DELETE	Interface
PARAMETER_INSERT	Interface
PARAMETER_ORDERING_CHANGE	Interface
PARAMETER_RENAMING	Interface
PARAMETER_TYPE_CHANGE	Interface
PARENT_CLASS_CHANGE	Interface
PARENT_CLASS_DELETE	Interface

Table A.1.: Mapping of the CTs by [268] that can be *directly* mapped onto the defect classes by [141].

Change Type (CT)	Defect Class
PARENT_CLASS_INSERT	Interface
PARENT_INTERFACE_CHANGE	Interface
PARENT_INTERFACE_DELETE	Interface
PARENT_INTERFACE_INSERT	Interface
REMOVED_CLASS	Interface
REMOVED_OBJECT_STATE	Data
REMOVING_ATTRIBUTE_MODIFIABILITY	Data
REMOVING_CLASS_DERIVABILITY	Interface
REMOVING_METHOD_OVERRIDABILITY	Interface
RETURN_TYPE_CHANGE	Interface
RETURN_TYPE_DELETE	Interface
RETURN_TYPE_INSERT	Interface

Table A.1.: Mapping of the CTs by [268] that can be *directly* mapped onto the defect classes by [141]. (Continued)

Condition	Defect Class
$CT \in \{$STATEMENT_*$\} \wedge$ $CE \in \{$ASSIGNMENT, POSTFIX_EXPRESSION, PREFIX_EXPRESSION$\} \wedge$ $PE \notin \{$FOR_INCR$\}$	Computation
$CT \in \{$STATEMENT_*$\} \wedge$ $CE \in \{$VARIABLE_DECLARATION_STATEMENT$\} \wedge$ $PE \notin \{$FOR_INIT$\}$	Data
$CT \in \{$UNCLASSIFIED_CHANGE$\} \wedge$ $CE \in \{$MODIFIER$\}$	Data
$CT \in \{$STATEMENT_*$\} \wedge$ $CE \in \{$METHOD_INVOCATION, CONSTRUCTOR_INVOCATION, SYNCHRONIZED_STATEMENT, CLASS_INSTANCE_CREATION$\}$	Interface
$CT \in \{$ADDING_FUNCTIONALITY, REMOVING_FUNCTIONALITY$\} \wedge$ $CE \in \{$METHOD$\}$	Interface
$CT \in \{$UNCLASSIFIED_CHANGE$\} \wedge$ $CE \in \{$TYPE_PARAMETER$\}$	Interface
$CT \in \{$STATEMENT_*$\} \wedge$ $CE \in \{$IF_STATEMENT, FOREACH_STATEMENT, CONTINUE_STATEMENT, RETURN_STATEMENT, THROW_STATEMENT, SWITCH_CASE, SWITCH_STATEMENT, BREAK_STATEMENT, CATCH_CLAUSE, TRY_STATEMENT, FOR_STATEMENT, WHILE_STATEMENT, DO_STATEMENT, LABELED_STATEMENT$\}$	Logic/Control
$CT \in \{$STATEMENT_*$\} \wedge$ $CE \in \{$ASSIGNMENT, POSTFIX_EXPRESSION, PREFIX_EXPRESSION$\} \wedge$ $PE \in \{$FOR_INCR$\}$	Logic/Control
$CT \in \{$STATEMENT_*$\} \wedge$ $CE \in \{$VARIABLE_DECLARATION_STATEMENT$\} \wedge$ $PE \in \{$FOR_INIT$\}$	Logic/Control
$CT \in \{$STATEMENT_*$\} \wedge$ $CE \in \{$ASSERT_STATEMENT$\}$	Other

Table A.2.: Mapping of the CTs by [268], where the CE and/or the PE needs to be taken into account to map a change onto the defect classes by [141]. The term STATEMENT_* includes the general change types, i.e., STATEMENT_UPDATE, STATEMENT_INSERT, STATEMENT_DELETE, STATEMENT_PARENT_CHANGE, STATEMENT_ORDERING_CHANGE.

B. Implementation Details

During our work, we developed several implementations. Within this section, we give more detailed information on the developed SmartSHARK plugins in Section B.1 and further details on the COMFORT framework in Section B.2.

B.1. SmartSHARK Plugins

During our work on SmartSHARK we and other colleagues developed several plugins that can be used within the SmartSHARK ecosystem. Table B.1 gives an overview of the currently available plugins of SmartSHARK, together with a short description of the plugin.

Name	Description
coastSHARK	Collects AST data from all files within the project
issueSHARK	Collects data from issue tracking systems
mailingSHARK	Collects data from project mailing lists
mecoSHARK	Collects metric and clone data
refSHARK	Detects refactorings of classes and methods
travisSHARK	Collects data from Travis [417]
vcsSHARK	Collects data from the version control system

Table B.1.: List of all data collection plugins of SmartSHARK.

B.2. COMFORT-Framework Implementations

The following tables highlight the different parts of the COMFORT-framework that we implemented. Table B.2 depicts all implemented loaders together with a short description, the language on which they can be applied, as well as the granularity that the output has (i.e., file, class, or method-level). Table B.3 show all implemented filters together with a short description and the input data that the filter can process. All implemented test metric collectors are highlighted in Table B.4. Besides the name of the collector, the table gives a short description for each collector, as well as the input data that it can process. Finally, Table B.5 shows all implemented filers together with a short description.

Name	Description	Languages	Granularity
CallGraph	Loads a static (Java) or dynamic (Python) call graph of the program	Java/Python	Method-Level
ChangeSet	Loads the change set (i.e., how often was the file changed together with a another one) for all files using the VCS of the project	Java/Python	File-Level
ClassFiles	Loads all *.class* files of the project	Java	File-Level
DependencyGraph	Loads a dependency graph (i.e., what units depend on which other units in the project) using xx (Java) or a script that extracts the imports to build a dependency graph (Python)	Java/Python	Class-Level
ProjectFiles	Loads all files of the project and separates them into test and production files	Java/Python	File-Level
TestCoverage	Loads the per-test coverage of the project	Java/Python	Method-Level

Table B.2.: List of all data loaders that are implemented within the COMFORT-Framework.

Name	Description	Input
DeletePythonPackages	Filters out python packages (i.e., *__init__.py* files)	Dependency graph (Python)
DirectConnectionToTest	Filters out all nodes that do not have a direct connection to a test	Call & Dependency graph
MergeInnerClassToMainClass	Merges the nodes that represent inner classes into their defining class by adding the corresponding edges to the main class node	Dependency graph (Java)
SameProject	Filter out all nodes that are not part of the project	Call & Dependency graph
TransformCallGraph	Transforms a call graph to a dependency graph	Call graph

Table B.3.: List of all filters that are implemented within the COMFORT-Framework.

Name	Description	Input
CoEvolutionTestType	Detects the test type by analyzing the change set: if a file is only changed together with one other file it is assumed to be a unit test and an integration test otherwise	Change set
Dependency	Calculates the number of dependent units of each test (i.e., how many units does the test need to work)	Call & Dependency graph, coverage
Directness	Calculates the percentage of units that are directly covered by the tests (i.e., units that have a direct connection in the call graph to the test)	Call graph)
CoveredLines	Calculates the TestLOC for each test	Coverage
IEEETestType	Classifies tests into unit and integration tests according to the IEEE definition	Call & Call & Dependency graph, coverage
ISTQBTestType	Classifies tests into unit and integration tests according to the ISTQB definition	Call & Dependency graph, coverage
LOCAndMcCabe	Calculates the LOC and McCabe complexity for tests	Project files
MaximumCallGraphDepth	Calculates the maximum depth of the call graph for each test	Call graph
MutationData	Calculates the mutation-detection capabilities of each test	Coverage (Java)
NamingConventionTestType	Classifies tests into unit and integration tests according to naming conventions	Project files, coverage
NumAssertion	Calculates the number of assertions of each test	Call graph, class files (Java)
TestCoverage	Calculates the percentage of covered units for each test	Call graph, dependency graph, coverage

Table B.4.: List of all metric collectors that are implemented within the COMFORT-Framework.

Name	Description
CSV	Stores the resulting data into a Comma Separated Values (CSV) file
SmartSHARK	Stores the resulting data into the SmartSHARK database

Table B.5.: List of all filers that are implemented within the COMFORT-Framework.

C. Test Statistics

C.1. Detailed Results for all Statistical Tests executed for the analysis of RQ 1

The following tables depict all concrete test statistics and p-values for all statistical tests that were performed in RQ1. Table C.1 depicts the input that was used for the Shapiro-Wilk tests, as well as the resulting test statistic including the p-value and the reference to the concrete research question. The same contents are shown for Table C.2 (results of the Brown-Forsythe tests) and Table C.3 (results for the Mann-Whitney U tests).

Input	Shapiro-Wilk Test Statistic	Reference to RQ
$NM_C(U_{IEEE})$	$(W = .8317, p = .0005)$	RQ 1.1
$NM_C(I_{IEEE})$	$(W = .9481, p = .1929)$	
$NM_C(U_{ISTQB})$	$(W = .8440, p = .0009)$	
$NM_C(I_{ISTQB})$	$(W = .9183, p = .0359)$	
$NM_TL(U_{IEEE})$	$(W = .7314, p = 1.0957 * 10^{-5})$	
$NM_TL(I_{IEEE})$	$(W = .4929, p = 1.4275 * 10^{-8})$	
$NM_TL(U_{ISTQB})$	$(W = .7960, p = .0001)$	
$NM_TL(I_{ISTQB})$	$(W = .5007, p = 1.7102 * 10^{-8})$	
$NM_C(U_{DEV})$	$(W = .8641, p = .0022)$	RQ 1.2
$NM_C(I_{DEV})$	$(W = .9466, p = .1774)$	
$NM_TL(U_{DEV})$	$(W = .5156, p = 2.4320 * 10^{-8})$	
$NM_TL(I_{DEV})$	$(W = .7492, p = 2.0327 * 10^{-5})$	

Table C.1.: Input and Shapiro-Wilk test statistic (including p-values) for all Shapiro-Wilk tests that were done to answer RQ1.

Input	Brown-Forsythe Test Statistic	Reference to RQ
$NM_C(U_{IEEE}), NM_C(I_{IEEE})$	$(F = .76, p = .3873)$	RQ 1.1
$NM_C(U_{ISTQB}), NM_C(I_{ISTQB})$	$(F = 7.1013, p = .0102)$	
$NM_TL(U_{IEEE}), NM_TL(I_{IEEE})$	$(F = 2.7057, p = .1060)$	
$NM_TL(U_{ISTQB}), NM_TL(I_{ISTQB})$	$(F = 4.4709, p = .0393)$	
$NM_C(U_{DEV}), NM_C(I_{DEV})$	$(F = 2.2856, p = .1366)$	RQ 1.2
$NM_TL(U_{DEV}), NM_TL(I_{DEV})$	$(F = .4733, p = .4945)$	

Table C.2.: Input and Brown-Forsythe test statistic (including p-values) for all Brown-Forsythe tests that were done to answer RQ1.

Input	Mann-Whitney-U Test Statistic	Reference to RQ
$NM_C(U_{IEEE}), NM_C(I_{IEEE})$	$(U = 149, p = 9.9810 * 10^{-5})$	RQ 1.1
$NM_C(U_{ISTQB}), NM_C(I_{ISTQB})$	$(U = 35, p = 6.2893 * 10^{-9})$	
$NM_TL(U_{IEEE}), NM_TL(I_{IEEE})$	$(U = 106, p = 4.0334 * 10^{-6})$	
$NM_TL(U_{ISTQB}), NM_TL(I_{ISTQB})$	$(U = 10, p = 4.5573 * 10^{-10})$	
$NM_C(U_{DEV}), NM_C(I_{DEV})$ (one-sided)	$(U = 388.5, p = .6642)$	RQ 1.2
$NM_C(U_{DEV}), NM_C(I_{DEV})$ (two-sided)	$(U = 340.5, p = .3422)$	
$NM_TL(U_{DEV}), NM_TL(I_{DEV})$ (one-sided)	$(U = 308.5, p = .1685)$	
$NM_TL(U_{DEV}), NM_TL(I_{DEV})$ (two-sided)	$(U = 308.5, p = .1685)$	

Table C.3.: Input and Mann-Whitney-U test statistic (including p-values) for all Mann-Whitney-U tests that were done to answer RQ1.

C.2. Detailed Results for all Statistical Tests executed for the analysis of RQ 2

The following tables depict all concrete test statistics and p-values for all statistical tests that were performed in RQ2. Table C.4 depicts the input that was used for the Shapiro-Wilk tests, as well as the resulting test statistic including the p-value and the reference to the concrete research question. The same contents are shown for Table C.5 (results of the Brown-Forsythe tests), Table C.6 (results for the Mann-Whitney U tests), and Table C.7 (results for the t-tests).

Input	Shapiro-Wilk Test Statistic	Reference to RQ
$RAT_{EXE}(U_{IEEE})$	$(W = .2636, p = 1.3389 * 10^{-10})$	RQ 2.1
$RAT_{EXE}(I_{IEEE})$	$(W = .7131, p = 5.9359 * 10^{-6})$	
$RAT_{EXE}(U_{ISTQB})$	$(W = .2683, p = 1.4556 * 10^{-10})$	
$RAT_{EXE}(I_{ISTQB})$	$(W = .7167, p = 6.6814 * 10^{-6})$	
$RAT_{EXE}(U_{DEV})$	$(W = .6508, p = 8.6040 * 10^{-7})$	
$RAT_{EXE}(I_{DEV})$	$(W = .6998, p = 3.8477 * 10^{-6})$	
$SCORE(U_{IEEE})$ (**ALL**)	$(W = .9233, p = .1681)$	RQ 2.2
$SCORE(U_{IEEE})$ (**ALL**, COMPUTATION)	$(W = .8442, p = .0088)$	
$SCORE(U_{IEEE})$ (**ALL**, DATA)	$(W = .8538, p = .0123)$	
$SCORE(U_{IEEE})$ (**ALL**, INTERFACE)	$(W = .9166, p = .1292)$	
$SCORE(U_{IEEE})$ (**ALL**, LOGIC/CONTROL)	$(W = .9318, p = .2334)$	
$SCORE(I_{IEEE})$ (**ALL**)	$(W = .9442, p = .3713)$	
$SCORE(I_{IEEE})$ (**ALL**, COMPUTATION)	$(W = .8116, p = .0029)$	
$SCORE(I_{IEEE})$ (**ALL**, DATA)	$(W = .8576, p = .0140)$	
$SCORE(I_{IEEE})$ (**ALL**, INTERFACE)	$(W = .9691, p = .8027)$	
$SCORE(I_{IEEE})$ (**ALL**, LOGIC/CONTROL)	$(W = .9253, p = .1816)$	
$SCORE(U_{IEEE})$ (**DISJ**)	$(W = .6392, p = 2.5598 * 10^{-5})$	
$SCORE(U_{IEEE})$ (**DISJ**, COMPUTATION)	$(W = .6935, p = 9.7608 * 10^{-5})$	
$SCORE(U_{IEEE})$ (**DISJ**, DATA)	$(W = .5400, p = 2.888 * 10^{-6})$	
$SCORE(U_{IEEE})$ (**DISJ**, INTERFACE)	$(W = .7003, p = 0.0001)$	
$SCORE(U_{IEEE})$ (**DISJ**, LOGIC/CONTROL)	$(W = .7607, p = .0006)$	
$SCORE(I_{IEEE})$ (**DISJ**)	$(W = .7743, p = .0010)$	
$SCORE(I_{IEEE})$ (**DISJ**, COMPUTATION)	$(W = .7031, p = .0001)$	
$SCORE(I_{IEEE})$ (**DISJ**, DATA)	$(W = .6596, p = 4.1783 * 10^{-5})$	
$SCORE(I_{IEEE})$ (**DISJ**, INTERFACE)	$(W = .7131, p = .0001)$	
$SCORE(I_{IEEE})$ (**DISJ**, LOGIC/CONTROL)	$(W = .7480, p = .0004)$	
$SCORE(U_{ISTQB})$ (**ALL**)	$(W = .8491, p = .0104)$	
$SCORE(U_{ISTQB})$ (**ALL**, COMPUTATION)	$(W = .7894, p = .0015)$	
$SCORE(U_{ISTQB})$ (**ALL**, DATA)	$(W = .8674, p = .0200)$	
$SCORE(U_{ISTQB})$ (**ALL**, INTERFACE)	$(W = .7908, p = .0015)$	
$SCORE(U_{ISTQB})$ (**ALL**, LOGIC/CONTROL)	$(W = .8323, p = .0058)$	
$SCORE(I_{ISTQB})$ (**ALL**)	$(W = .9408, p = .3281)$	
$SCORE(I_{ISTQB})$ (**ALL**, COMPUTATION)	$(W = .8806, p = .0325)$	
$SCORE(I_{ISTQB})$ (**ALL**, DATA)	$(W = .8572, p = .0138)$	
$SCORE(I_{ISTQB})$ (**ALL**, INTERFACE)	$(W = .9730, p = .8674)$	
$SCORE(I_{ISTQB})$ (**ALL**, LOGIC/CONTROL)	$(W = .9297, p = .2154)$	

$SCORE(U_{ISTQB})$ (**DISJ**)	$(W = .7682, p = .0008)$	
$SCORE(U_{ISTQB})$ (**DISJ**, COMPUTATION)	$(W = .6430, p = 2.2794 * 10^{-5})$	
$SCORE(U_{ISTQB})$ (**DISJ**, DATA)	$(W = .6171, p = 1.5302 * 10^{-5})$	
$SCORE(U_{ISTQB})$ (**DISJ**, INTERFACE)	$(W = .7466, p = .0004)$	
$SCORE(U_{ISTQB})$ (**DISJ**, LOGIC/CONTROL)	$(W = .8391, p = .0073)$	
$SCORE(I_{ISTQB})$ (**DISJ**)	$(W = .7222, p = .0002)$	
$SCORE(I_{ISTQB})$ (**DISJ**, COMPUTATION)	$(W = .6728, p = 5.7750 * 10^{-5})$	
$SCORE(I_{ISTQB})$ (**DISJ**, DATA)	$(W = .5754, p = 6.0739 * 10^{-6})$	
$SCORE(I_{ISTQB})$ (**DISJ**, INTERFACE)	$(W = .8710, p = .0228)$	
$SCORE(I_{ISTQB})$ (**DISJ**, LOGIC/CONTROL)	$(W = .6952, p = .0001)$	
$SCORE(U_{DEV})$ (**ALL**)	$(W = .9531, p = .5068)$	
$SCORE(U_{DEV})$ (**ALL**, COMPUTATION)	$(W = .8837, p = .0365)$	
$SCORE(U_{DEV})$ (**ALL**, DATA)	$(W = .9169, p = .1311)$	
$SCORE(U_{DEV})$ (**ALL**, INTERFACE)	$(W = .9643, p = .7125)$	
$SCORE(U_{DEV})$ (**ALL**, LOGIC/CONTROL)	$(W = .9469, p = .4091)$	**RQ 2.2**
$SCORE(I_{DEV})$ (**ALL**)	$(W = .8734, p = .0249)$	
$SCORE(I_{DEV})$ (**ALL**, COMPUTATION)	$(W = .7493, p = .0004)$	
$SCORE(I_{DEV})$ (**ALL**, DATA)	$(W = .7382, p = .0003)$	
$SCORE(I_{DEV})$ (**ALL**, INTERFACE)	$(W = .8991, p = .0656)$	
$SCORE(I_{DEV})$ (**ALL**, LOGIC/CONTROL)	$(W = .8728, p = .0244)$	
$SCORE(U_{DEV})$ (**DISJ**)	$(W = .6964, p = .0001)$	
$SCORE(U_{DEV})$ (**DISJ**, COMPUTATION)	$(W = .7771, p = .0009)$	
$SCORE(U_{DEV})$ (**DISJ**, DATA)	$(W = .6509, p = 3.3850 * 10^{-5})$	
$SCORE(U_{DEV})$ (**DISJ**, INTERFACE)	$(W = .7281, p = .0002)$	
$SCORE(U_{DEV})$ (**DISJ**, LOGIC/CONTROL)	$(W = .6381, p = 2.4945 * 10^{-5})$	
$SCORE(I_{DEV})$ (**DISJ**)	$(W = .8493, p = .0105)$	
$SCORE(I_{DEV})$ (**DISJ**, COMPUTATION)	$(W = .6747, p = 6.0610 * 10^{-5})$	
$SCORE(I_{DEV})$ (**DISJ**, DATA)	$(W = .6562, p = 3.8396 * 10^{-5})$	
$SCORE(I_{DEV})$ (**DISJ**, INTERFACE)	$(W = .5598, p = 4.3536 * 10^{-6})$	
$SCORE(I_{DEV})$ (**DISJ**, LOGIC/CONTROL)	$(W = .7884, p = .0014)$	
$DL_{AVG}(U_{IEEE})$ (**ALL**)	$(W = .7810, p = .0011)$	
$DL_{AVG}(I_{IEEE})$ (**ALL**)	$(W = .8937, p = .0533)$	
$DL_{AVG}(U_{ISTQB})$ (**ALL**)	$(W = .9560, p = .5584)$	
$DL_{AVG}(I_{ISTQB})$ (**ALL**)	$(W = .9111, p = .1042)$	
$DL_{AVG}(U_{DEV})$ (**ALL**)	$(W = .8502, p = .0108)$	
$DL_{AVG}(I_{DEV})$ (**ALL**)	$(W = .8622, p = .0165)$	
$DL_{AVG}(U_{IEEE})$ (**DISJ**)	$(W = .8138, p = .0032)$	**RQ 2.3**
$DL_{AVG}(I_{IEEE})$ (**DISJ**)	$(W = .9117, p = .1067)$	
$DL_{AVG}(U_{ISTQB})$ (**DISJ**)	$(W = .9491, p = .4420)$	
$DL_{AVG}(I_{ISTQB})$ (**DISJ**)	$(W = .9270, p = .1935)$	
$DL_{AVG}(U_{DEV})$ (**DISJ**)	$(W = .9298, p = .2161)$	
$DL_{AVG}(I_{DEV})$ (**DISJ**)	$(W = .8993, p = .0660)$	

Table C.4.: Input and Shapiro-Wilk test statistic (including p-values) for all Shapiro-Wilk tests that were done to answer RQ2.

Input	Brown-Forsythe Test Statistic	Reference to RQ
$RAT_{EXE}(U_{IEEE})$, $RAT_{EXE}(I_{IEEE})$	$(F = 0.5772, p = .4508)$	
$RAT_{EXE}(U_{ISTQB})$, $RAT_{EXE}(I_{ISTQB})$	$(F = 0.6958, p = .4080)$	**RQ 2.1**
$RAT_{EXE}(U_{DEV})$, $RAT_{EXE}(I_{DEV})$	$(F = 0.0088, p = .9258)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**ALL**)	$(F = .7817, p = .3832)$	**RQ 2.2**
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**ALL**, COMPUTATION)	$(F = .0002, p = .9895)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**ALL**, DATA)	$(F = .0532, p = .8191)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**ALL**, INTERFACE)	$(F = .7310, p = .3989)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**ALL**, LOGIC/CONTROL)	$(F = .4490, p = .5076)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**)	$(F = 2.2217, p = .1359)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**, COMPUTATION)	$(F = 0.9716, p = .3317)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**, DATA)	$(F = 1.2642, p = .2692)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**, INTERFACE)	$(F = 1.8549, p = .1827)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**, LOGIC/CONTROL)	$(F = 3.9303, p = .0561)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**ALL**)	$(F = 2.0864, p = .1583)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**ALL**, COMPUTATION)	$(F = 1.2007, p = .2184)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**ALL**, DATA)	$(F = .7603, p = .3897)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**ALL**, INTERFACE)	$(F = 4.1370, p = .0503)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**ALL**, LOGIC/CONTROL)	$(F = 1.0049, p = .3236)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**DISJ**)	$(F = 1.2655, p = .0470)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**DISJ**, COMPUTATION)	$(F = 1.2109, p = .2794)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**DISJ**, DATA)	$(F = 2.0214, p = .1648)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**DISJ**, INTERFACE)	$(F = 4.0008, p = .0540)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**DISJ**, LOGIC/CONTROL)	$(F = 8.6605, p = .0060)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**ALL**)	$(F = 0.2975, p = .5892)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**ALL**, COMPUTATION)	$(F = .0225, p = .8818)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**ALL**, DATA)	$(F = .2663, p = .6094)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**ALL**, INTERFACE)	$(F = .9199, p = .3447)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**ALL**, LOGIC/CONTROL)	$(F = .2175, p = .6441)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**)	$(F = 2.7982, p = .1041)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**, COMPUTATION)	$(F = 5.1429, p = .0302)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**, DATA)	$(F = .6939, p = .4110)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**, INTERFACE)	$(F = 1.3571, p = .2527)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**, LOGIC/CONTROL)	$(F = 2.4625, p = .1264)$	
$DL_{AVG}(U_{IEEE})$, $DL_{AVG}(I_{IEEE})$ (**ALL**)	$(F = 5.8472, p = .0215)$	**RQ 2.3**
$DL_{AVG}(U_{ISTQB})$, $DL_{AVG}(I_{ISTQB})$ (**ALL**)	$(F = 14.7811, p = .0005)$	
$DL_{AVG}(U_{DEV})$, $DL_{AVG}(I_{DEV})$ (**ALL**)	$(F = 3.8220, p = .0594)$	
$DL_{AVG}(U_{IEEE})$, $DL_{AVG}(I_{IEEE})$ (**DISJ**)	$(F = 4.9191, p = .0338)$	
$DL_{AVG}(U_{ISTQB})$, $DL_{AVG}(I_{ISTQB})$ (**DISJ**)	$(F = 12.6466, p = .0012)$	
$DL_{AVG}(U_{DEV})$, $DL_{AVG}(I_{DEV})$ (**DISJ**)	$(F = 5.4284, p = .0263)$	

Table C.5.: Input and Brown-Forsythe test statistic (including p-values) for all Brown-Forsythe tests that were done to answer RQ2.

Input	Mann-Whitney-U Test Statistic	Reference to RQ
$RAT_{EXE}(U_{IEEE})$, $RAT_{EXE}(I_{IEEE})$	$(U = 340, p = .3390)$	**RQ 2.1**
$RAT_{EXE}(U_{ISTQB})$, $RAT_{EXE}(I_{ISTQB})$	$(U = 338, p = .3264)$	
$RAT_{EXE}(U_{DEV})$, $RAT_{EXE}(I_{DEV})$	$(U = 341, p = .3453)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**)	$(U = 102, p = .0740)$	**RQ 2.2**
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**, COMPUTATION)	$(U = 114.5, p = .1517)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**, DATA)	$(U = 122, p = .2204)$	
$SCORE(U_{IEEE})$, $SCORE(I_{IEEE})$ (**DISJ**, INTERFACE)	$(U = 127, p = .2782)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**DISJ**, COMPUTATION)	$(U = 136.5, p = .3974)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**DISJ**, DATA)	$(U = 106, p = .0915)$	
$SCORE(U_{ISTQB})$, $SCORE(I_{ISTQB})$ (**DISJ**, LOGIC/CONTROL)	$(U = 128, p = .2905)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**)	$(U = 117, p = .1761)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**, COMPUTATION)	$(U = 104, p = .0704)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**, DATA)	$(U = 136.5, p = .3925)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**, INTERFACE)	$(U = 127.5, p = .2837)$	
$SCORE(U_{DEV})$, $SCORE(I_{DEV})$ (**DISJ**, LOGIC/CONTROL)	$(U = 106, p = .1264)$	
$DL_{AVG}(U_{IEEE})$, $DL_{AVG}(I_{IEEE})$ (**ALL**)	$(U = 30, p = 4.3084 * 10^{-5})$	**RQ 2.3**
$DL_{AVG}(U_{DEV})$, $DL_{AVG}(I_{DEV})$ (**ALL**)	$(U = 85, p = .0211)$	
$DL_{AVG}(U_{IEEE})$, $DL_{AVG}(I_{IEEE})$ (**DISJ**)	$(U = 24, p = 1.7886 * 10^{-5})$	
$DL_{AVG}(U_{DEV})$, $DL_{AVG}(I_{DEV})$ (**DISJ**)	$(U = 78, p = .0115$	

Table C.6.: Input and Mann-Whitney-U test statistic (including p-values) for all Mann-Whitney-U tests that were done to answer RQ2.

Input	T-Test Statistic	Reference to RQ
$SCORE(U_{IEEE}), SCORE(I_{IEEE})$ (**ALL**)	$(T = -.7931, p = .4335)$	**RQ 2.2**
$SCORE(U_{IEEE}), SCORE(I_{IEEE})$ (**ALL**, COMPUTATION)	$(T = -.2119, p = .8335)$	
$SCORE(U_{IEEE}), SCORE(I_{IEEE})$ (**ALL**, DATA)	$(T = .2998, p = .7663)$	
$SCORE(U_{IEEE}), SCORE(I_{IEEE})$ (**ALL**, INTERFACE)	$(T = -1.6332, p = .1122)$	
$SCORE(U_{IEEE}), SCORE(I_{IEEE})$ (**ALL**, LOGIC/CONTROL)	$(T = -.6330, p = .5312)$	
$SCORE(U_{IEEE}), SCORE(I_{IEEE})$ (**DISJ**, LOGIC/CONTROL)	$(T = 2.0049, p = .0535)$	
$SCORE(U_{ISTQB}), SCORE(I_{ISTQB})$ (**ALL**)	$(T = -2.0265, p = .0511)$	
$SCORE(U_{ISTQB}), SCORE(I_{ISTQB})$ (**ALL**, COMPUTATION)	$(T = -1.5626, p = .1280)$	
$SCORE(U_{ISTQB}), SCORE(I_{ISTQB})$ (**ALL**, DATA)	$(T = -.8300, p = .4127)$	
$SCORE(U_{ISTQB}), SCORE(I_{ISTQB})$ (**ALL**, INTERFACE)	$(T = -3.0590, p = .0045)$	
$SCORE(U_{ISTQB}), SCORE(I_{ISTQB})$ (**ALL**, LOGIC/CONTROL)	$(T = -1.6541, p = .1079)$	
$SCORE(U_{ISTQB}), SCORE(I_{ISTQB})$ (**DISJ**)	$(T = 1.6772, p = .1032)$	
$SCORE(U_{ISTQB}), SCORE(I_{ISTQB})$ (**DISJ**, INTERFACE)	$(T = 1.2083, p = .2358)$	
$SCORE(U_{DEV}), SCORE(I_{DEV})$ (**ALL**)	$(T = -0.1311, p = .8965)$	
$SCORE(U_{DEV}), SCORE(I_{DEV})$ (**ALL**, COMPUTATION)	$(T = .3773, p = .7085)$	
$SCORE(U_{DEV}), SCORE(I_{DEV})$ (**ALL**, DATA)	$(T = -.1573, p = .8760)$	
$SCORE(U_{DEV}), SCORE(I_{DEV})$ (**ALL**, INTERFACE)	$(T = -.4957, p = .6235)$	
$SCORE(U_{DEV}), SCORE(I_{DEV})$ (**ALL**, LOGIC/CONTROL)	$(T = .0116, p = .9908)$	
$DL_{AVG}(U_{ISTQB})$ and $DL_{AVG}(I_{ISTQB})$ (**ALL**)	$(T = -5.9298, p = 1.7852 * 10^{-5})$	**RQ 2.3**
$DL_{AVG}(U_{ISTQB})$ and $DL_{AVG}(I_{ISTQB})$ (**DISJ**)	$(T = -6.4520, p = 6.4940 * 10^{-6})$	

Table C.7.: Input and T-test statistic (including p-values) for all T-tests that were done to answer RQ2.

D. Additional Data for RQ 2.2

Within this section, we present additional data for RQ 2.2. The following tables show the number of detected defects (separated by defect type) for unit and integration tests. Furthermore, these tables highlight the number of defects that were detected by both test types. Within this section, we present four different tables for the **ALL** and **DISJ** data sets, as well as the **IEEE** and **ISTQB** definitions and the developer classification, which we used to classify our tests into unit and integration tests. Furthermore, we show additional visualizations. On the one hand, we show different box plots that highlight and aggregated view on the number of detected defects (overall and defect-type specific). On the other hand, we show Venn-diagrams for each project that visualizes this data on a project level.

D.1. Tables of the Killed Mutants per Defect Type

For each data set (i.e., **ALL** and **DISJ**) there exist three different tables, depicting the concrete numbers of killed mutants per defect type separated by unit and integration tests, and mutants that are killed by both. Table D.1 shows these numbers for the **ALL** data set and unit and integration tests that are classified via the IEEE definition, Table D.2 shows them for the ISTQB definition, and Table D.3 for the developer classification.

Table D.4 depicts the number of killed mutants per defect type for unit and integration tests as classified by the IEEE definition, Table D.5 shows them for the ISTQB definition, and Table D.6 for the developer classification.

	COMP.			DATA			INT.			L/C		
Project	UT	IT	B	UT	IT	B	UT	IT	B	UT	IT	B
commons-beanutils	42	102	41	69	147	51	421	1444	308	760	1989	601
commons-codec	439	25	136	828	75	191	1555	391	223	2397	510	637
commons-collections	141	325	143	188	290	122	524	1839	413	1266	3555	1018
commons-io	191	199	98	349	255	136	1005	854	438	1763	1375	881
commons-lang	903	185	215	2253	482	391	4599	1763	1211	13009	2996	2720
commons-math	307	6108	847	453	12075	1659	936	20497	1767	2110	33961	5078
druid	78	4113	36	99	3996	40	114	28309	99	327	35867	311
fastjson	62	2085	279	131	4530	532	124	8829	499	519	13586	1698
google-gson	145	190	193	129	158	278	143	1285	291	381	1747	1016
guice	3	513	25	5	420	39	60	3029	107	96	3582	189
HikariCP	11	179	12	19	168	11	44	735	44	32	954	53
jackson-core	43	2689	173	67	2711	161	144	3433	143	275	10364	474
jfreechart	96	2389	59	188	2903	108	199	9060	115	745	18538	714
joda-time	21	1152	42	56	1835	101	183	8456	199	245	13305	714
jsoup	4	382	65	14	692	103	29	3300	267	81	4380	595
mybatis-3	23	397	321	27	350	190	280	3405	1843	296	3321	2277
zxing	65	1130	192	298	2920	469	318	3900	438	681	8439	1331
Mean	151.41	1303.71	169.24	304.29	2000.41	269.53	628.12	5913.47	494.41	1469.59	9321.71	1194.53
StDev	226.13	1696.41	197.39	543.79	2991.41	388.96	1104.77	7619.44	560.70	3059.68	10957.89	1227.60

Table D.1.: Number of mutations for the **ALL** data set that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) separated by defect type. The tests are classified according to the **IEEE** definition.

Project	COMP.			DATA			INT.			L/C		
	UT	IT	B	UT	IT	B	UT	IT	B	UT	IT	B
commons-beanutils	6	175	4	11	226	30	28	2080	65	65	3088	197
commons-codec	73	504	23	206	840	48	400	1439	330	686	2585	273
commons-collections	88	427	94	123	386	91	288	2281	207	797	4442	600
commons-io	128	304	56	146	524	70	272	1842	183	484	3139	396
commons-lang	621	497	185	1532	1195	399	2707	3811	1055	8906	7160	2659
commons-math	44	6810	408	73	13193	921	165	22438	597	481	38200	2468
druid	51	4154	22	67	4049	19	59	28425	38	202	36090	213
fastjson	20	2310	96	38	4996	159	22	9281	149	206	15012	585
google-gson	17	327	184	39	262	264	56	1384	279	116	2068	960
guice	0	537	4	0	460	4	4	3184	8	6	3847	14
HikariCP	5	188	9	8	181	9	20	777	26	13	993	33
jackson-core	23	2720	162	52	2744	143	103	3494	123	192	10533	388
jfreechart	57	2429	58	133	2964	102	134	9132	108	537	18778	682
joda-time	1	1207	7	7	1963	22	15	8795	28	46	14053	165
jsoup	3	403	45	9	721	79	18	3413	165	38	4530	488
mybatis-3	8	711	22	8	531	28	46	5235	247	37	5502	355
zxing	16	1239	132	51	3341	295	50	4444	162	137	9710	604
Mean	68.29	1467.18	88.88	147.24	2269.18	157.82	258.06	6556.18	221.76	761.71	10572.35	651.76
StDev	146.89	1777.80	103.73	361.73	3191.08	226.55	641.38	7668.09	258.25	2113.57	11208.73	760.44

Table D.2.: Number of mutations for the **ALL** data set that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) separated by defect type. The tests are classified according to the **ISTQB** definition.

Project	COMP.			DATA			INT.			L/C		
	UT	IT	B	UT	IT	B	UT	IT	B	UT	IT	B
commons-beanutils	0	185	0	0	267	0	0	2173	0	0	3350	0
commons-codec	378	35	187	686	129	279	1450	244	475	2255	332	957
commons-collections	533	10	66	542	1	57	2468	71	237	5192	118	529
commons-io	236	207	45	284	389	67	686	1387	227	1052	2441	528
commons-lang	1071	85	148	2274	425	442	5314	1216	1161	14182	2321	2351
commons-math	4588	403	2271	9330	1234	3623	16244	1967	4989	26330	2734	12085
druid	284	2858	1085	244	2865	1026	980	21803	5739	1368	26083	9054
fastjson	33	1829	564	58	3854	1281	240	6621	2591	538	10971	4294
google-gson	144	129	255	159	123	283	321	839	559	624	1135	1385
guice	25	134	382	34	168	262	165	1159	1872	279	1392	2196
HikariCP	27	53	122	41	56	101	116	350	357	107	398	534
jackson-core	211	2125	569	209	2213	517	359	2639	722	882	8054	2177
jfreechart	2273	14	257	2988	33	178	7708	219	1447	17916	214	1867
joda-time	221	263	731	505	641	846	1773	4018	3047	3406	4918	5940
jsoup	61	139	251	175	204	430	971	871	1754	1025	1189	2842
mybatis-3	51	123	567	58	124	385	640	1001	3887	672	998	4224
zxing	23	1192	172	125	3084	478	167	3941	548	329	8874	1248
Mean	597.59	575.53	451.29	1041.88	930.00	603.24	2329.53	2971.71	1741.88	4479.82	4442.47	3071.24
StDev	1172.05	873.50	548.74	2291.00	1255.61	853.29	4136.30	5146.34	1749.45	7599.29	6466.21	3270.33

Table D.3.: Number of mutations for the **ALL** data set that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) separated by defect type. The tests are classified according to the developer classification.

Project	COMP.			DATA			INT.			L/C		
	UT	IT	B	UT	IT	B	UT	IT	B	UT	IT	B
commons-beanutils	1	0	4	1	1	1	2	4	12	6	17	58
commons-codec	6	2	6	6	0	0	16	3	7	44	2	34
commons-collections	0	3	5	4	0	1	9	9	23	22	29	62
commons-io	4	2	14	7	2	6	23	2	43	29	9	81
commons-lang	20	1	22	38	3	22	89	5	40	287	3	132
commons-math	2	48	50	1	25	71	4	27	56	20	161	334
druid	2	47	4	2	23	3	4	117	5	10	201	34
fastjson	0	42	47	0	50	51	0	105	51	4	246	227
google-gson	2	0	4	0	3	4	0	2	4	2	9	12
guice	0	5	4	0	6	2	0	15	11	1	24	11
HikariCP	0	0	2	0	1	0	0	1	5	0	1	4
jackson-core	0	6	10	0	1	2	1	4	3	5	13	42
jfreechart	4	11	15	2	0	8	3	17	9	13	41	46
joda-time	0	9	5	0	14	8	2	50	11	4	222	99
jsoup	0	0	5	0	1	4	0	1	15	0	1	15
mybatis-3	0	4	5	0	0	5	3	1	53	1	2	86
zxing	1	1	5	0	2	5	0	0	3	1	4	28
Mean	2.47	10.65	12.18	3.59	7.76	11.35	9.18	21.35	20.65	26.41	57.94	76.76
StDev	4.86	17.04	14.63	9.14	13.44	19.68	21.53	36.11	19.53	68.26	87.61	86.16

Table D.4.: Number of mutations for the **DISJ** data set that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) separated by defect type. The tests are classified according to the **IEEE** definition.

Project	COMP.			DATA			INT.			L/C		
	UT	IT	B	UT	IT	B	UT	IT	B	UT	IT	B
commons-beanutils	1	3	1	0	3	0	0	16	2	1	60	20
commons-codec	2	9	3	5	0	1	5	8	13	23	16	41
commons-collections	0	4	4	4	0	1	4	15	22	17	50	46
commons-io	1	9	10	5	5	5	5	40	23	14	54	51
commons-lang	15	7	21	30	9	24	51	37	46	228	38	156
commons-math	1	63	36	0	36	61	3	47	37	17	250	248
druid	2	47	4	2	25	1	4	119	3	9	204	32
fastjson	0	77	12	0	76	25	0	134	22	4	394	79
google-gson	0	2	4	0	5	2	0	3	3	1	11	11
guice	0	7	2	0	6	2	0	23	3	0	34	2
HikariCP	0	0	2	0	1	0	0	4	2	0	2	3
jackson-core	0	7	9	0	1	2	1	4	3	2	26	32
jfreechart	4	11	15	2	1	7	2	23	4	12	48	40
joda-time	0	11	3	0	17	5	2	60	1	4	279	42
jsoup	0	2	3	0	3	2	0	5	11	0	6	10
mybatis-3	0	5	4	0	2	3	2	34	21	1	39	49
zxing	1	3	3	0	2	5	0	1	2	0	6	27
Mean	1.59	15.71	8.00	2.82	11.29	8.59	4.65	33.71	12.82	19.59	89.24	52.29
StDev	3.62	23.09	9.04	7.24	19.39	15.45	12.09	39.06	13.65	54.21	116.83	61.83

Table D.5.: Number of mutations for the **DISJ** data set that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) separated by defect type. The tests are classified according to the **ISTQB** definition.

Project	COMP.			DATA			INT.			L/C		
	UT	IT	B	UT	IT	B	UT	IT	B	UT	IT	B
commons-beanutils	0	5	0	0	3	0	0	18	0	0	81	0
commons-codec	9	0	5	5	0	1	16	2	8	70	0	10
commons-collections	5	0	3	5	0	0	25	0	16	77	3	33
commons-io	6	9	5	7	4	4	29	20	19	24	48	47
commons-lang	35	0	8	46	2	15	99	13	22	321	18	83
commons-math	21	0	79	17	0	80	26	2	59	123	4	388
druid	2	3	48	2	2	24	4	12	110	9	18	218
fastjson	0	5	84	0	9	92	0	15	141	2	46	429
google-gson	1	0	5	0	1	6	0	1	5	2	2	19
guice	0	1	8	0	1	7	0	2	24	1	3	32
HikariCP	0	0	2	0	1	0	1	1	4	0	1	4
jackson-core	1	3	12	0	0	3	1	2	5	3	16	41
jfreechart	25	0	5	9	1	0	23	0	6	84	0	16
joda-time	1	0	13	1	0	21	5	1	57	25	1	299
jsoup	0	0	5	0	0	5	2	1	13	1	0	15
mybatis-3	0	0	9	0	0	5	3	0	54	2	0	87
zxing	0	5	2	0	5	2	0	3	0	0	22	11
Mean	6.24	1.82	17.24	5.41	1.71	15.59	13.76	5.47	31.94	43.76	15.47	101.88
StDev	10.54	2.72	26.48	11.43	2.42	27.55	24.45	6.99	40.54	80.93	22.83	140.67

Table D.6.: Number of mutations for the **DISJ** data set that are killed by Unit Tests (UT), Integration Tests (IT), and Both (B) separated by defect type. The tests are classified according to the developer classification.

D.2. Box Plots of Defect Detection Scores

Within this section, several summary statistics in form of Box-plots are given for RQ 2.2. Figure D.1 depicts the scores of the **ALL** and **DISJ** data sets for unit and integration tests according to the IEEE and ISTQB definitions and the developer classification. Figures D.2, D.3, and D.4 show the scores for the **ALL** and **DISJ** data sets for unit and integration tests separated by defect type for the IEEE and ISTQB definitions, and the developer classification, respectively.

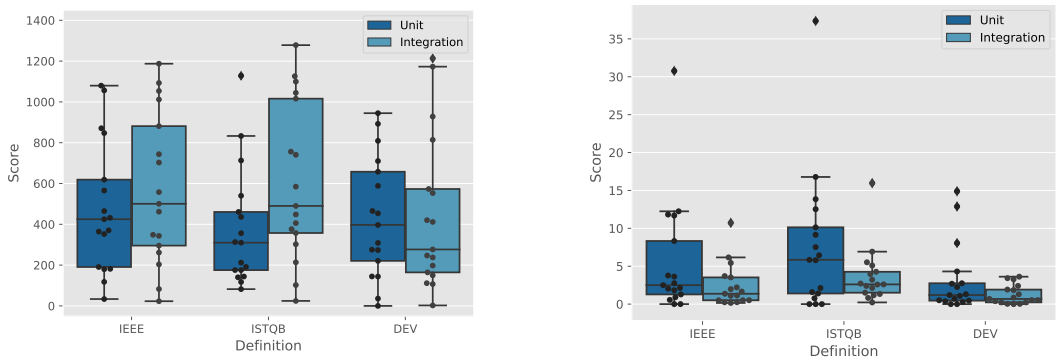


Figure D.1.: Box plots of the scores for the **ALL** (left) and **DISJ** (right) data sets for unit and integration tests according to the **IEEE** and **ISTQB** definitions and the developer classification. The points in the plot represent the concrete values for each project.

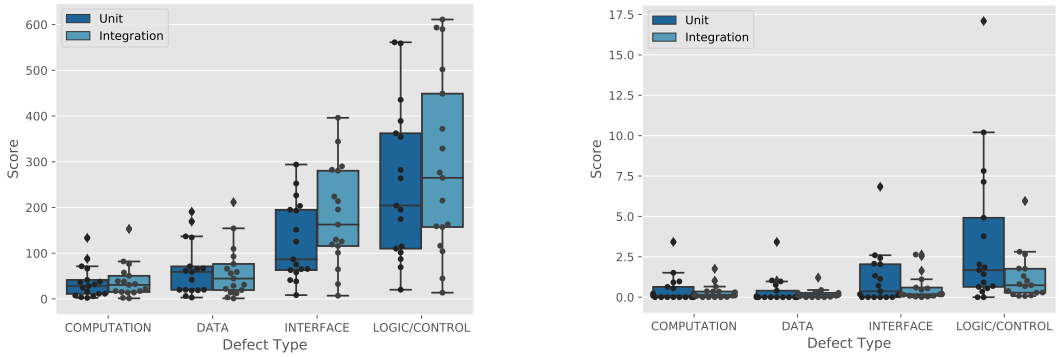


Figure D.2.: Box plots of the scores for the **ALL** (left) and **DISJ** (right) data sets for unit and integration tests separated by defect type for the **IEEE** definition. The points in the plot represent the concrete values for each project.

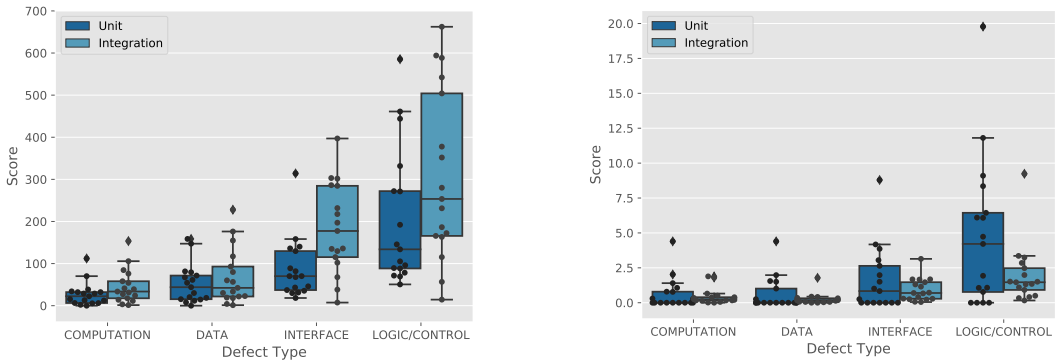


Figure D.3.: Box plots of the scores for the **ALL** (left) and **DISJ** (right) data sets for unit and integration tests separated by defect type for the **ISTQB** definition. The points in the plot represent the concrete values for each project.

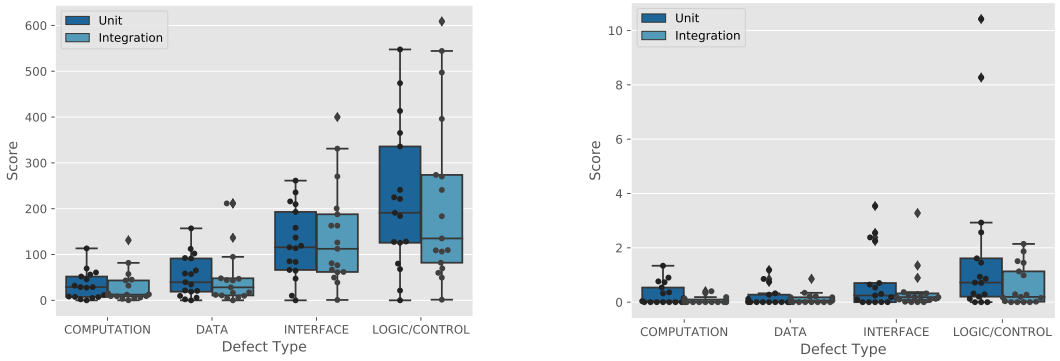
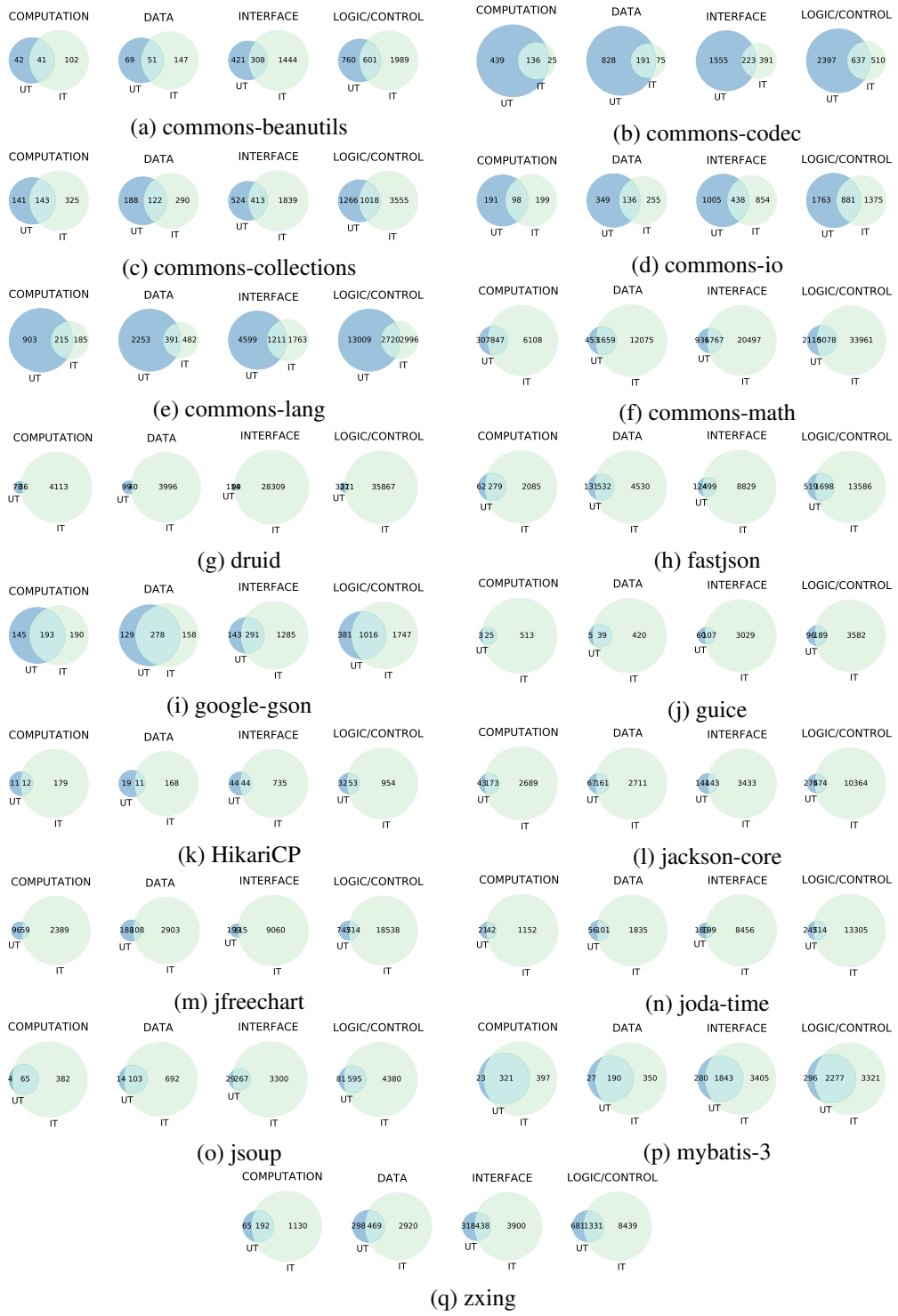


Figure D.4.: Box plots of the scores for the **ALL** (left) and **DISJ** (right) data sets for unit and integration tests separated by defect type for the developer classification. The points in the plot represent the concrete values for each project.

D.3. Venn-Diagrams

Within this section we visualize the data shown in Section D.1. The Venn-diagrams show the number of killed mutants for the **ALL** and **DISJ** data sets for unit and integration tests of each project. Figure D.5 depicts these numbers for the **ALL** data set and for tests that are classified according to the IEEE definition. Figure D.6 shows the same data, but tests are classified according to the ISTQB definition. The same goes for Figure D.7. Within this figure, the tests are classified according to the developer classification. Figures D.8, D.9, and D.10 depict Venn-diagrams showing the number of killed mutants for the **DISJ** data sets of unit and integration tests for each project, where the tests are classified according to the IEEE and ISTQB definition and according to the developer of the project, respectively.



(a) commons-beanutils

(b) commons-codec

(c) commons-collections

(d) commons-io

(e) commons-lang

(f) commons-math

(g) druid

(h) fastjson

(i) google-gson

(j) guice

(k) HikariCP

(l) jackson-core

(m) jfreechart

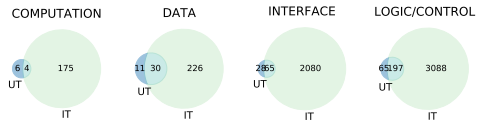
(n) joda-time

(o) jsoup

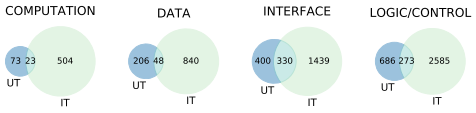
(p) mybatis-3

(q) zxing

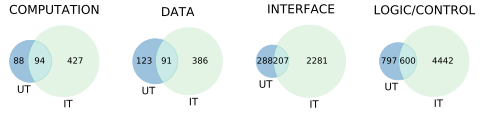
Figure D.5.: Venn-diagrams showing the number of mutations for the **ALL** data set that are killed by Unit Tests (UT) and Integration tests (IT) together with their intersection, separated by defect type. The tests are classified according to the **IEEE** definition.



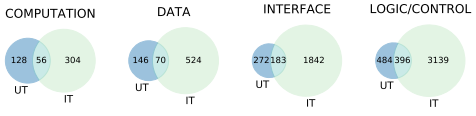
(a) commons-beautils



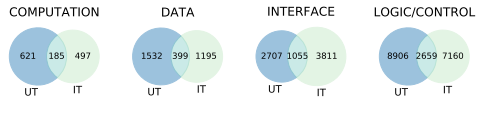
(b) commons-codec



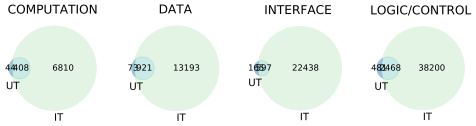
(c) commons-collections



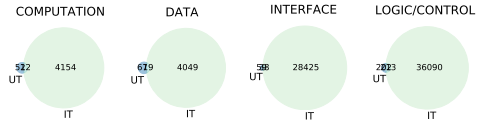
(d) commons-io



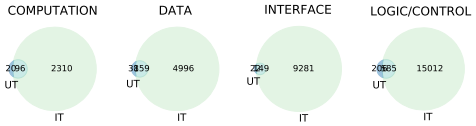
(e) commons-lang



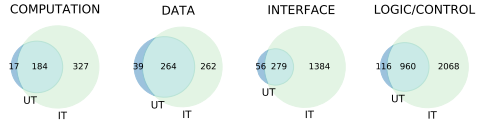
(f) commons-math



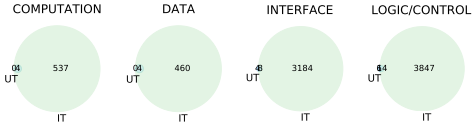
(g) druid



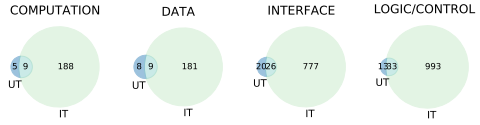
(h) fastjson



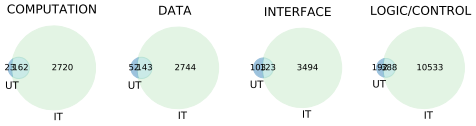
(i) google-gson



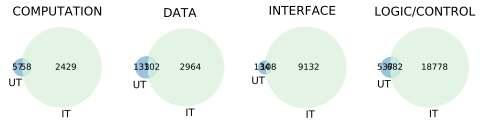
(j) guice



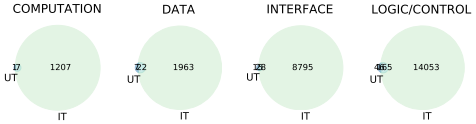
(k) HikariCP



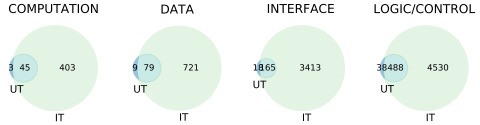
(l) jackson-core



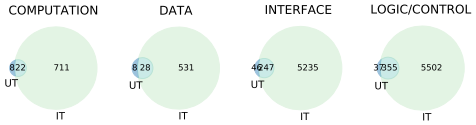
(m) jfreechart



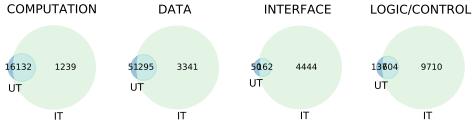
(n) joda-time



(o) jsoup

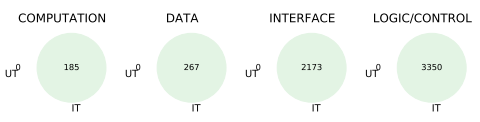


(p) mybatis-3

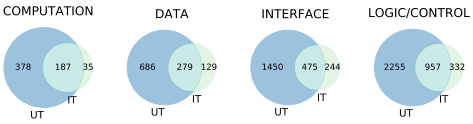


(q) zxing

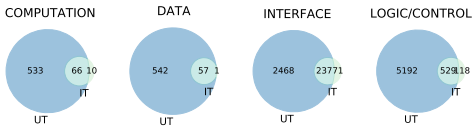
Figure D.6.: Venn-diagrams showing the number of mutations for the **ALL** data set that are killed by Unit Tests (UT) and Integration tests (IT) together with their intersection, separated by defect type. The tests are classified according to the **ISTQB** definition.



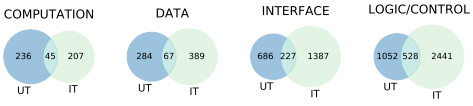
(a) commons-beanutils



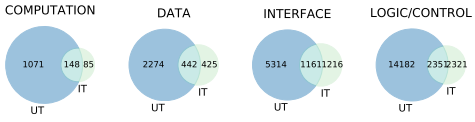
(b) commons-codec



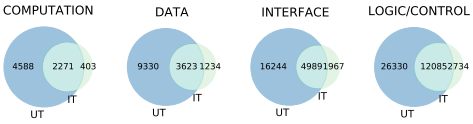
(c) commons-collections



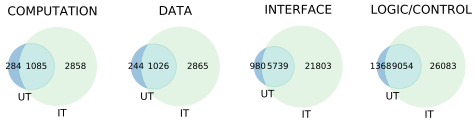
(d) commons-io



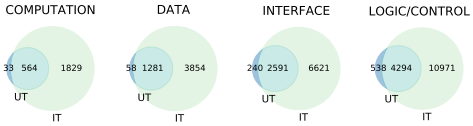
(e) commons-lang



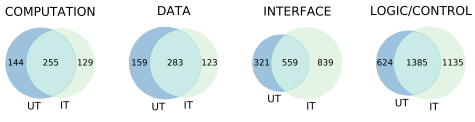
(f) commons-math



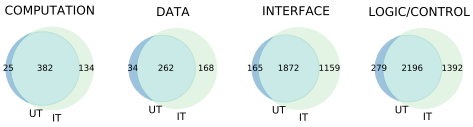
(g) druid



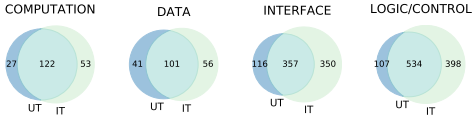
(h) fastjson



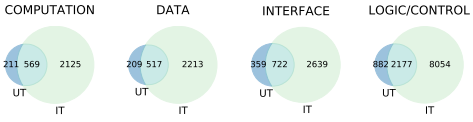
(i) google-gson



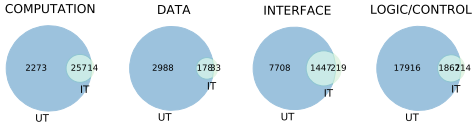
(j) guice



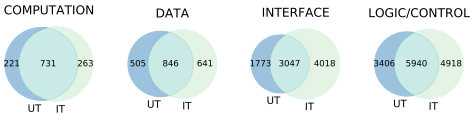
(k) HikariCP



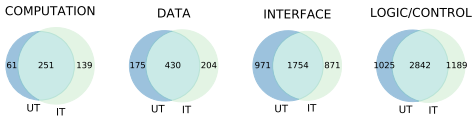
(l) jackson-core



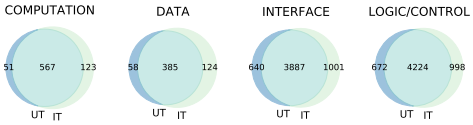
(m) jfreechart



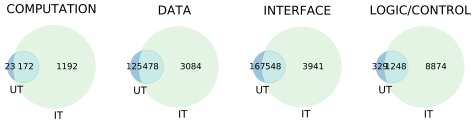
(n) joda-time



(o) jsoup

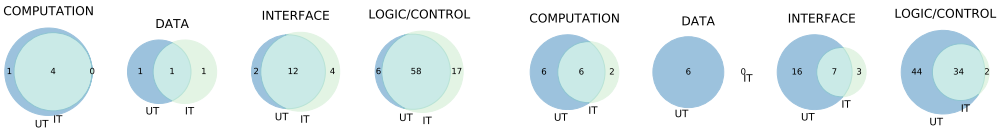


(p) mybatis-3



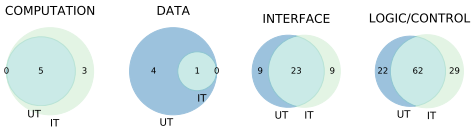
(q) zxing

Figure D.7.: Venn-diagrams showing the number of mutations for the **ALL** data set that are killed by Unit Tests (UT) and Integration tests (IT) together with their intersection, separated by defect type. The tests are classified according to the developer classification.

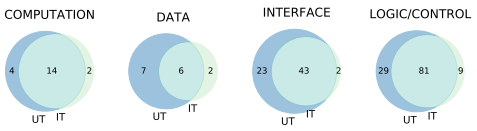


(a) commons-beanutils

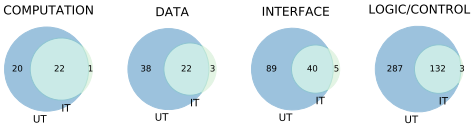
(b) commons-codec



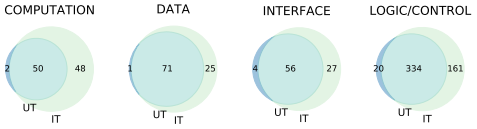
(c) commons-collections



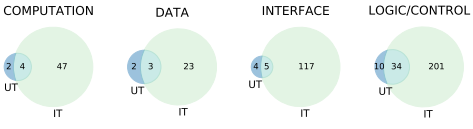
(d) commons-io



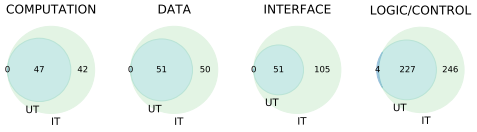
(e) commons-lang



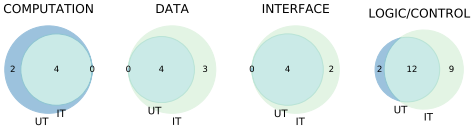
(f) commons-math



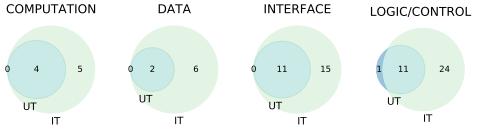
(g) druid



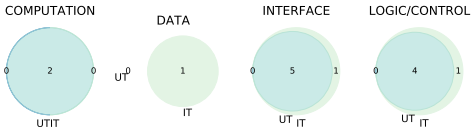
(h) fastjson



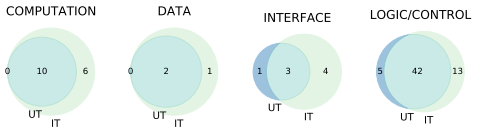
(i) google-gson



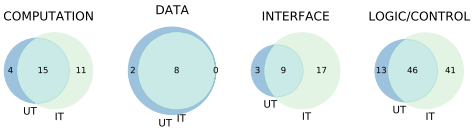
(j) guice



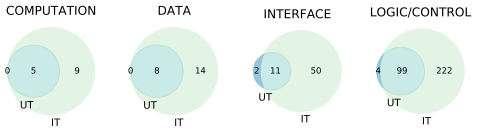
(k) HikariCP



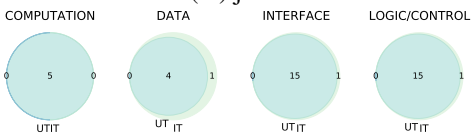
(l) jackson-core



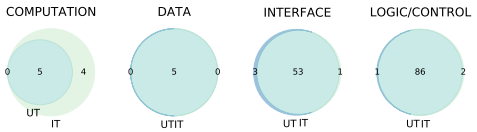
(m) jfreechart



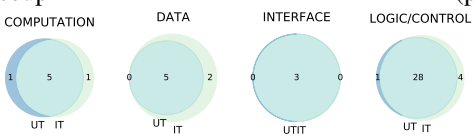
(n) joda-time



(o) jsoup

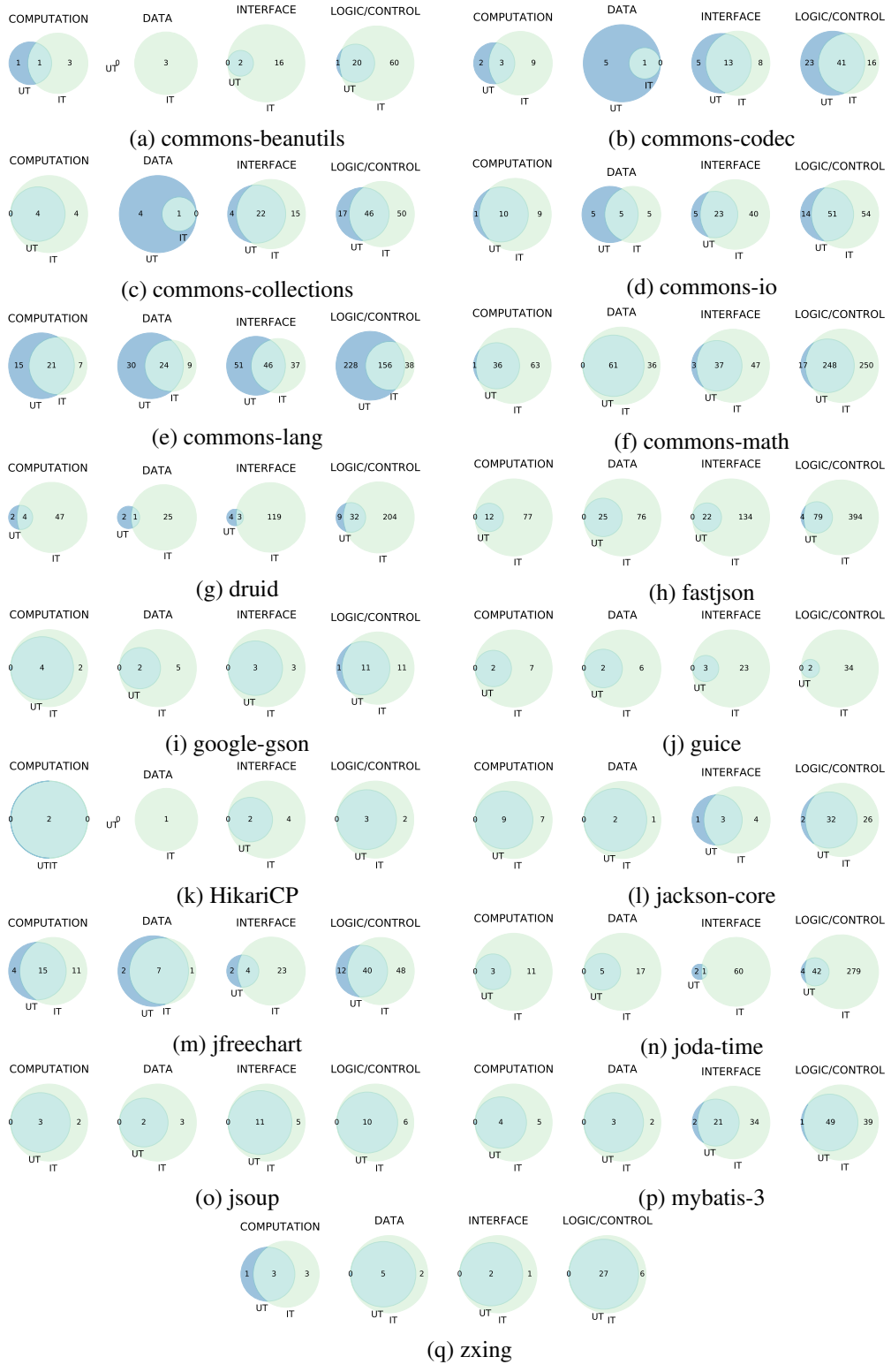


(p) mybatis-3



(q) zxing

Figure D.8.: Venn-diagrams showing the number of mutations for the **DISJ** data set that are killed by Unit Tests (UT) and Integration tests (IT) together with their intersection, separated by defect type. The tests are classified according to the **IEEE** definition.



(a) commons-beanutils

(b) commons-codec

(c) commons-collections

(d) commons-io

(e) commons-lang

(f) commons-math

(g) druid

(h) fastjson

(i) google-gson

(j) guice

(k) HikariCP

(l) jackson-core

(m) jfreechart

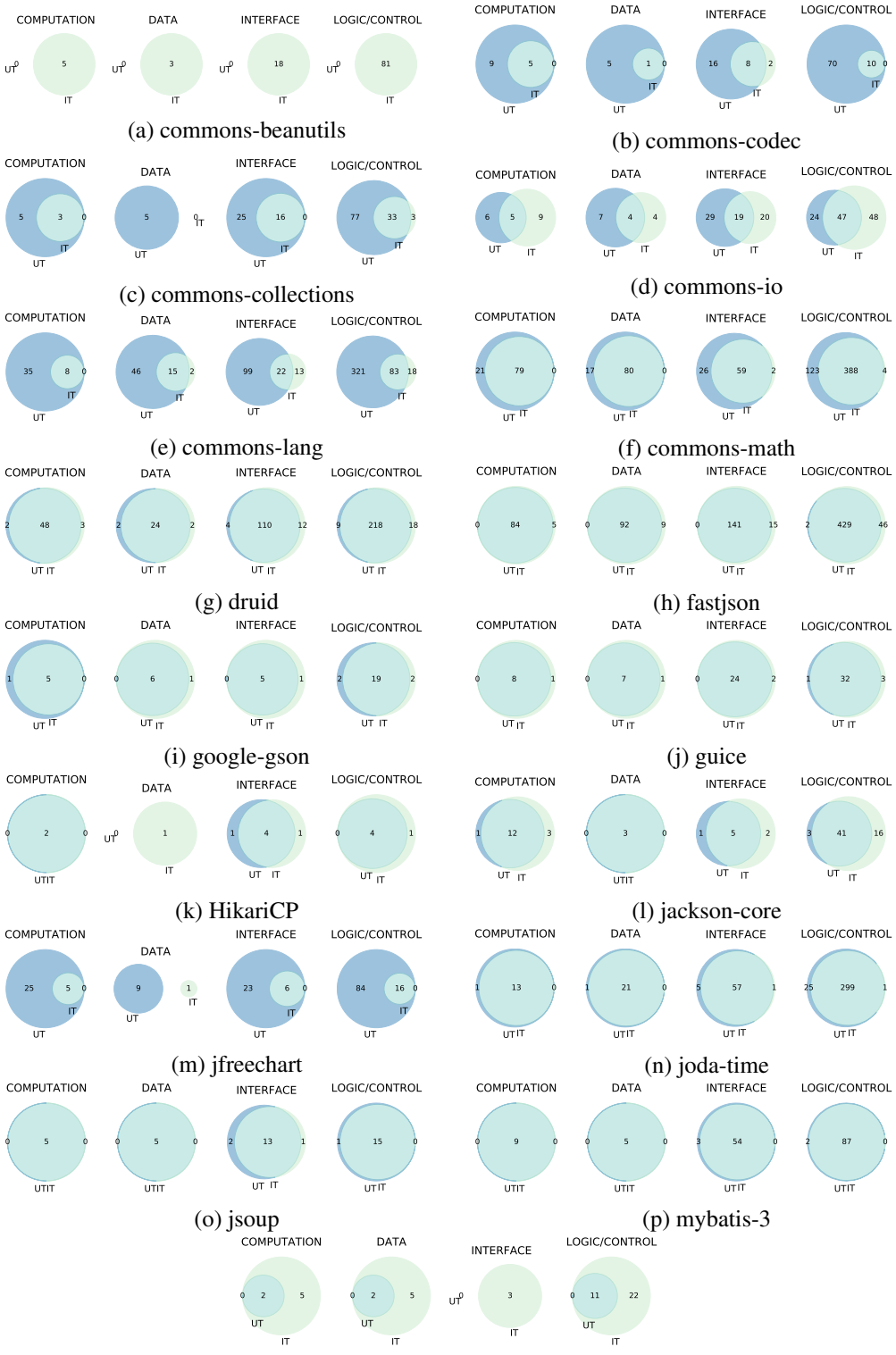
(n) joda-time

(o) jsoup

(p) mybatis-3

(q) zxing

Figure D.9.: Venn-diagrams showing the number of mutations for the **DISJ** data set that are killed by Unit Tests (UT) and Integration tests (IT) together with their intersection, separated by defect type. The tests are classified according to the **ISTQB** definition.



(a) commons-beanutils

(b) commons-codec

(c) commons-collections

(d) commons-io

(e) commons-lang

(f) commons-math

(g) druid

(h) fastjson

(i) google-gson

(j) guice

(k) HikariCP

(l) jackson-core

(m) jfreechart

(n) joda-time

(o) jsoup

(p) mybatis-3

(q) zxing

Figure D.10.: Venn-diagrams showing the number of mutations for the **DISJ** data set that are killed by Unit Tests (UT) and Integration tests (IT) together with their intersection, separated by defect type. The tests are classified according to developer classification.