Understanding Test Code Quality from Perspective of Test Code Design and Maintenance

Dong Jae Kim

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Computer Science) at

Concordia University

Montréal, Québec, Canada

August 2024

© Dong Jae Kim, 2024

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Dong Jae Kim

Entitled: Understanding Test Code Quality from Perspective of Test Code Design and Maintenance

and submitted in partial fulfillment of the requirements for the degree of

Doctor Of Philosophy Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Chair
Dr. Ahmed Soliman

External Examiner
Dr. Andy Zaidman

Arm's Length Examiner
Dr. Yann-Gaël Guéhéneuc

Examiner
Dr. Diego Elias Costa

Examiner
Dr. Peter C. Rigby

Thesis Supervisor (s)
Dr. Tse-Hsun (Peter) Chen

Approved by
Dr. Leila Kosseim Chair of Department or Graduate Program Director

August 19, 2024
Date of Defence

Dr. Mourad Debbabi Dean, Faculty of Engineering and Computer Science

Abstract

Understanding Test Code Quality from Perspective of Test Code Design and Maintenance

Dong Jae Kim

Software testing is vital for ensuring software reliability and robustness. It involves executing a program and verifying it against developer-defined criteria to identify and fix deviations, aiming for fault-free software. Despite extensive research on automated test prioritization, fault localization, and program repair, test design and maintenance remain under-explored. This dissertation aims to understand test code quality from the perspective of design and maintenance practices, exploring various aspects of test design through manual classification, quantitative methods, and automated approaches.

The first part of the dissertation examines test smells, which are design issues that impact test code comprehension and maintainability. Although widely accepted in academia, it is unclear whether developers address test smells in practice. The thesis investigates developers' awareness of test smells and their impact on defect-proneness. Findings reveal that many proposed test smells persist and are removed incidentally through code deletion and refactoring, with minimal effect on software defect density. This dissertation aims to provide empirical support for re-ranking current test smell catalogs.

As test automation grows, modern frameworks like JUnit and TestNG are increasingly used in Java-based systems. These frameworks introduce annotation-driven development, which manages crucial aspects of the test execution lifecycle. Our second aim is to use mining software repository techniques to provide insights into how modern testing frameworks are used to improve test code design and maintenance. First, we investigate annotation usage in test code, providing a manual classification of their API usage and misuses (e.g.,

test smells). Second, we conduct a detailed empirical analysis of one controversial practice: disabling tests using JUnit and TestNG's @Ignore annotation. While this alleviates maintenance challenges, it introduces technical debt as it does not validate software quality. Finally, we examine the tradeoffs between reusability and redundancy in test code practices, particularly through inheritance, highlighting issues in test increasing test execution time.

Through case studies and experiments, the techniques proposed in this dissertation offer new insights into test code quality that may guide the development of automated tool support in the future, which may help developers improve test maintenance.

Acknowledgments

I want to thank my supervisor, Dr. Tse-Hsun (Peter) Chen, for his ongoing support throughout my doctoral studies. Without his supervision and resources, this thesis would not have been possible, and more importantly, I would not have had the opportunity to become the researcher I am today.

I also want to thank Dr. Jinqiu Yang for her supervision and for bringing a new perspective to the research direction. I greatly value her rigor in research thinking and constructive criticism. I also want to thank Dr. Nikolaos Tsantalis for his supervision. I was fortunate to have the opportunity to work with him and learn from the refactoring master.

I also want to thank my committee members, Professor Peter Rigby, Professor Diego Elias Costa, and Professor Yann-Gaël Guéhéneuc, for their supervision and mentorship. I am grateful to my external committee member, Andy Zaidman, for his insights and inputs that have significantly improved this thesis. The discussions we had in the defense have helped me think about future research directions.

I want to thank my colleagues in the SPEAR lab and Concordia's Department of Software Engineering. Specifically, I want to thank PengZi for recommending me to the SPEAR lab, which has opened tremendous opportunities for me.

I want to thank my parents for their support and belief in me when I decided to go back to school to pursue a doctoral degree. I am also deeply thankful to my wife, Yanning Liu, for always being there for me and supporting me through challenging times.

Dedication

To my parents, my friends, and my wife, Yanning Liu.

Related Publications

In all chapters and related publications of the thesis, my contributions are: drafting the initial research idea; researching background knowledge and related work; implementing the tools; conducting experiments; and writing and polishing the writing. My co-authors supported me in refining the initial ideas, pointing me to missing related work, providing feedback on earlier drafts, and polishing the writing.

Earlier versions of the work in the thesis were published as listed below:

1. *The Secret Life of Test Smells - An Empirical Study on Test Smell Evolution and Maintenance.*
   Kim, Dong Jae and Chen, Tse-Hsun and Yang, Jinqiu. Empirical Software Engineering Journal (**EMSE 2021**). 34 pages. (*Chapter 3*)

2. *Studying Test Annotation Maintenance in the Wild.*
   Kim, Dong Jae and Tsantalis, Nikolaos and Chen, Tse-Hsun and Yang, Jinqiu. The 43rd IEEE/ACM International Conference on Software Engineering (**ICSE 2021**). 12 pages. (*Chapter 4*)

3. *How Disabled Tests Manifest in Test Maintainability Challenges?*
   Kim, Dong Jae and Yang, Bo and Yang, Jinqiu and Chen, Tse-Hsun. The 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (**ESEC/FSE 2021**). 12 pages. (*Chapter 5*)

4. *A First Look at the Inheritance-Induced Redundant Test Execution*
   Kim, Dong Jae and Yang, Jinqiu and Chen, Tse-Hsun.

The 46th International Conference on Software Engineering (**ICSE-2024**). 12 pages. (*Chapter 6*)

During his Ph.D studies, the author also contributed to the following publications that do not directly relate to the core topic of this thesis:

1. *Towards Better Graph Neural Neural Network-based Fault Localization Through Enhanced Code Representation.*
   Nakhla Rafi, Md and Kim, Dong Jae and Chen, An Ran and Chen, Tse-Hsun and Wang, Shaowei. International Conference on the Foundations of Software Engineering (**FSE-2024**). 10 pages.

2. *Decoding Anomalies! Unraveling Operational Challenges in Human-in-the-Loop Anomaly Validation.*
   Kim, Dong Jae and Chen, Tse-Hsun. International Conference on the Foundations of Software Engineering (**FSE-SEIP-2024**). 5 pages.

3. *LLMParser: An Exploratory Study on Using Large Language Models for Log Parsing.*
   Ma, Zeyang and Chen, An Ran and Kim, Dong Jae and Chen, Tse-Hsun and Wang, Shaowei. 46th International Conference on Software Engineering (**ICSE-2024**). 10 pages.

4. *Challenges in Adopting Artificial Intelligence Based User Input Verification Framework in Reporting Software Systems.*
   Kim, Dong Jae and Locke, Steve and Chen, Tse-Hsun Peter and Toma, Andrei and Sporea, Steve and Weinkam, Laura and Sajedi, Sarah. 45th International Conference on Software Engineering: Software Engineering in Practice (**ICSE-SEIP-2023**). 10 pages.

5. *An empirical study on performance bugs in deep learning frameworks.*
   Makkouk, Tarek and Kim, Dong Jae and Chen, Tse-Hsun Peter. 2022 International Conference on Software Maintenance and Evolution (**ICSME-2022**). 10 pages.

Contents

Related Publications **vii**

List of Figures **xiv**

List of Tables **xvi**

1 Introduction **1**

1.1 Thesis Goal 1

1.2 Thesis Objective 2

1.3 Thesis Contribution 6

1.3.1 Research Contribution for Objective 1 6

1.3.2 Research Contribution for Objective 2 6

1.3.3 Research Contribution for Objective 3 7

1.4 Thesis Overview 8

2 Background **9**

2.1 Context of the work 9

2.1.1 Empirical Software Engineering 9

2.1.2 Automated Software Testing 10

2.2 Test Quality 10

2.2.1 Code Coverage 11

2.2.2 Limitation of Common Test Quality Attribute 11

2.3 Test Quality from Perspective of Test Maintainability, Extensibility and Reusability 12
2.3.1 Test Smells 12
2.4 Modern Java-Based Testing Framework 14
2.4.1 Junit4 Test Automation 15
2.5 Fine-Grain Code Differencing for Mining Developers' Code Change History 16
2.5.1 Mining Code Change History 16

3 Literature Review 18

3.1 Paper Selection Process 19
3.2 Empirical studies on practitioners perceptions' on test smells in real-world systems 19
3.2.1 Perception and Maintenance of Test Smells 20
3.2.2 Using design issue as metric to understand software post-release defect 21
3.3 Research on improving test maintainability practices from perspective of test automation 22
3.3.1 Improving test maintainability from framework usages 22
3.3.2 Empirical study on technical debt from the perspective of test code 23

4 Demystifying test smells and their misconceptions about their harmfulness in test maintainability 25

4.1 Introduction 26
4.2 Background 28
4.2.1 Brief Overview of Test Smells 28
4.2.2 Identifying Test Smells 29
4.3 Studied Systems 30
4.4 Results 32
4.4.1 RQ1: How do test smells evolve overtime? 32
4.4.2 RQ2: What is the motivation behind removing test smells? 39
4.4.3 RQ3: What is the relationship between test smells and software quality? 52

4.5 Threats to Validity 60
4.6 Implication & Contribution 63
4.7 Chapter Summary 64

5 Demystifying test annotation maintenance in the wild **65**
5.1 Introduction 66
5.2 Method for Detecting Changes in Test Annotations 69
5.3 Studied Systems 72
5.4 Results 74
5.4.1 RQ1: How common are test annotation changes? 74
5.4.2 RQ2: How are test annotations changed in the wild? 75
5.4.3 RQ3: Why do developers change test annotations? 78
5.5 Threats to Validity 88
5.6 Implication & Contribution 88
5.6.1 Discussion and Implication for Researchers 88
5.6.2 Discussion and Implication for Application Developers and Testers . 90
5.6.3 Discussion and Implication for Framework Designers 91
5.7 Chapter Summary 91

6 Demystifying test disabling practices to improve test maintainability **93**
6.1 Introduction 94
6.2 Method for Detecting Test Disabling Practice 97
6.2.1 Definition of Test Disabling Practice 97
6.2.2 Detecting Disabled and Active Tests 98
6.2.3 Tracking the Evolution of Disabled Tests 100
6.2.4 Studied Systems 102
6.3 Results 103
6.3.1 RQ1: How Common are Test Disabling Changes? 103
6.3.2 RQ2: What is the Change Pattern of Disabled Tests? 105
6.3.3 RQ3: Why do developers utilize test disabling practice? 108

6.4 Threats to Validty 114
6.5 Discussion and Implication 116
6.5.1 Discussion and Implication for Researchers 116
6.5.2 Discussion and Implication for Developers 118
6.6 Chapter Summary 118

7 Demystifying inheritance in test code and impact on test redundancies 120
7.1 Introduction 121
7.2 Motivation 123
7.3 Related Works 126
7.4 Technique for Identifying Inheritance-Induced Redundant Test Executions . 128
7.4.1 Statically Detecting *Redundancy-Inducing Inheritance* 128
7.4.2 Statically Detecting *Inheritance-Induced Redundant Test Candidates* 131
7.4.3 Detecting *Inheritance-Induced Redundant Test Executions* through Dynamic Analysis 132
7.5 Results 135
7.5.1 RQ1: How Prevalent are Inheritance-Induced Redundant Test Candidates? 136
7.5.2 RQ2: Are the Inheritance-Induced Redundant Test Candidates Truly Redundant? 139
7.5.3 RQ3: How Much Time does Inheritance Induced Redundant Test Contribute to the Overall Test Execution Time? 142
7.5.4 RQ4: Assessing the Feasibility of Reducing Inheritance-Induced Redundant Test Execution? 144
7.6 Threats to Validity 148
7.7 Implications & Contribution 150
7.7.1 Discussion and Implication for Researchers. 150
7.7.2 Discussion and Implication for Practitioners. 152
7.8 Chapter Summary 152

8 Conclusion and Future Work **153**

8.1 Thesis Contribution 153

8.2 Future Work 155

Appendix A Supplementary Figure **158**

Bibliography **167**

List of Figures

Figure 1.1 An overall view of this thesis 4

Figure 4.1 Time series plots that show the evolution of the test smell density (normalized average) of the studied systems. The test smell densities are calculated based on seven snapshots that are taken every six months between 2016 and 2019. 34

Figure 4.2 Time series plots that show the evolution of the averaged raw test smell of the studied systems. The test smells are aggregated from on seven snapshots taken every six months between 2016 and 2019. 35

Figure 4.3 Assertion refactoring, removing duplicate assertion and Assertion Roulette test smells. Located in the commit 3b42fb5 from Apache Karaf. 47

Figure 4.4 Conditional logic refactoring, removing complex conditional logic using assertions. Located in the commit 7ebc5da6 from Apache Kafka. 48

Figure 7.1 Developers re-use test through inheritance to ease maintenance. . . . 125

Figure 7.2 Overview of the End-to-End Process for Finding Redundant Test Cases.129

Figure 7.3 Overlap between Redundant and Non-redundant Tests in Test Execution. The overlapping region (orange) indicates that an inherited test case is redundant in one `subclass` but non-redundant in another `subclass`. The one figures on the left correspond to five systems, while the two figures on the right correspond to the rest. 145

Figure 7.4 Analysis of distance in the inheritance tree between the parent test case and the child test. 147

List of Tables

Table 4.1 An overview of the studied systems. 31
Table 4.2 An overview of the developer experience and the number of contributors in the studied systems. 32
Table 4.3 The comparison of the test smell density (number of test smell instances per 1000 lines of test code) for each type of test smell in the studied systems from 2016 and 2019. 36
Table 4.4 The comparison of the prevalence of test smells (i.e., the raw number of test smell instances) for the studied systems from 2016 to 2019. 37
Table 4.5 A summary of our manual classification on the commits that removes test smell, i.e., 304 sampled commits minus 12 commits that are incorrectly flagged by the test smell detection tool. Our analysis focuses on the context of each commit and how the test smell is addressed. In particular, we show the association between test code changes and the corresponding maintenance activities that developers apply. 42
Table 4.6 The comparison of the area under (a ROC) curve for the studied systems. The model is trained using the system in the first column, and AUC is calculated using the system depicted in the remaining columns. 59
Table 4.7 Summary of our findings and their implications. 63
Table 5.1 A brief overview on commonly used JUnit4 annotations. 67
Table 5.2 Precision and recall of our extended version of RefactoringMiner. . . 71
Table 5.3 An overview of the studied systems (from 2015 to 2020). 72

Table 5.4 Use of Annotations from Different Frameworks in Test Code Released in 2015 and 2020, respectively. 73
Table 5.5 Mined Test Code Changes in 82,810 Commits. 74
Table 5.6 Quantitative Analysis: Top three highest frequency of annotation addition, removal and modification. 76
Table 5.7 Quantitative Analysis of annotation replacements. 76
Table 5.8 manual classification: Taxonomy of Annotation Changes. 81
Table 6.1 Precision of test tracking, i.e., detecting the three types of commit changes, including disabling a test, re-enabling a test, and deleting a disabled test. 101
Table 6.2 An overview of the studied systems (from 2015 to 2020). 102
Table 6.3 Frequency comparison between test disabling changes and the common test code refactorings at the same program element level. 104
Table 6.4 The frequency of various types of test disabling-related changes. *Disabling Tests* shows the total number of changes that disable tests. *Re-enabled Tests* shows the total number of changes that re-enable tests and whether developers simultaneously modified the tests (i.e., modified vs. unmodified). *Deleting Disabled Tests* shows the number of changes that delete disabled tests. 104
Table 6.5 Change patterns of disabled tests. Unresolved tests represent the tests that remain disabled at the end of the studied period. Resolved tests represent the tests the are either re-enabled or deleted completely at the end of the studied period. 106
Table 6.6 The distributions of the average time (in days) for a disabled test to become re-enabled, deleted, or remain disabled (i.e., developers did not modify the test in the studied period after it was re-enabled). 107
Table 6.7 manual classification result: a taxonomy of why developers disable tests. 110

Table 6.8 An example of an invalid block comment that cannot be detected by the tool. Invalid comment represents a mix between natural language and code. 115

Table 7.1 Analyzed repository characteristics. 124

Table 7.2 Systems Studied and Their Inheritance Statistics. 135

Table 7.3 Revealing the landscape of redundant test candidates: A summary of the static analysis result. 138

Table 7.4 The prevalence of redundant tests and their impact on test execution time. *Candidates* refers to *Inheritance-Induced Redundant Test Candidates* and *Redundant* refers to true redundant test cases. 143

Table A.1 The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric. 159

Table A.2 The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric. 160

Table A.3 The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric. 161

Table A.4 The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric. 162

Table A.5 The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric. 163

Table A.6 The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric. 164

Table A.7 The effect size of the test smell metrics on post-release defects. Effect is measured by setting the subject metric to 110 % and 150 % of it mean value, while other metrics are kept at their mean values. Bolded numbers indicate a positive increase in effect. 165

Table A.8 The effect size of the test smell metrics on post-release defects. Effect is measured by setting the subject metric to 110 % and 150 % of it mean value, while other metrics are kept at their mean values. Bolded numbers indicate a positive increase in effect. 166

Table A.9 The effect size of the test smell metrics on post-release defects. Effect is measured by setting the subject metric to 110 % and 150 % of it mean value, while other metrics are kept at their mean values. Bolded numbers indicate a positive increase in effect. 166

Chapter 1

Introduction

Figure 1.1 provides a summary of the roadmap of our thesis. Specifically, this chapter introduces the statement of our thesis, starting with our goal, followed by the thesis objectives, and concluding with our thesis outcomes and contributions.

1.1 Thesis Goal

Software testing is pivotal for ensuring the reliability of software applications. With advancements in technology, there has been an increased research effort aimed at automating the tedious nature of testing, considering various aspects of the testing process. One of the most important forms of test automation is regression testing, where tests must be run after every code change to detect bugs and ensure that the software is fault-free. To improve test automation to minimize testing cost, plethora of research investigates following aspect of test automation: (1) prioritizing test cases to be run first out of a tremendous number of tests to quickly detect faults, (2) localizing the bug once it is uncovered, and (3) repairing the buggy code to allow the test to pass.

As software evolves, certain tests may become obsolete or require modifications to align with updated functionalities. Test may also require good design decisions and principals to enable ease of extensibility and maintainability. While advancements in test automation have led to significant improvements to uncover faults quickly and minimize software

costs, improving test code design and maintainability remains an under-explored area, and researchers overlook the challenges associated with maintaining test suites over time. The software industry requires robust standards for test design and maintenance, which can significantly impact overall software quality. Therefore, this thesis aims to investigate and mitigate the challenges of test code maintainability, particularly focusing on manual classification, quantitative and automated approaches to improve maintainability in Java-based software systems. In particular, the thesis seeks to offer practical insights into enhancing test code design and maintenance practices for better software quality assurance practices.

1.2 Thesis Objective

In our attempt to improve test code design, we adopt an alternative perspective, which is to uncover and resolve design issues that may hinder test code maintainability. Hence, the main research objective for this thesis work is:

> Given the critical role of test code in ensuring software reliability, it is imperative to address design issues that affect its long-term maintainability. Surprisingly, many prior studies have overlooked these issues, focusing primarily on software test automation. Therefore, there is an empirical need to investigate design practices in test code to offer insights that can benefit the software industry.

Based on this objective, our primary research questions are:

(1) What is the current state of knowledge about design issues in test code, particularly regarding their applicability to industrial settings? To what extent do developers prioritize identifying and resolving these design issues to improve software quality?

(2) As test automation tools become more prevalent in test code design, how do these tools impact the presence and resolution of design issues in test code? Are there established design practices available for developers using such tools to optimize the resolution of these issues?

(3) How does the use of reusability in test code design affect the presence and resolution of design issues? What is the impact of test reusability on the effectiveness of detecting bugs and maintaining long-term code quality?

This thesis takes one first step towards improving test quality from the perspective of test design and maintainability. In particular, based on key questions above, the thesis proposes three major aims to improve test design as listed below:

Objective 1 (O1): Although academic research has identified numerous test design issues, their practical applicability remains unclear. *Our objective is to re-evaluate current knowledge of test design issues and analyze their applicability in software industries.* Similar to source code, test code can also have design issues that negatively impact its maintainability. These design issues are commonly referred to as "*test smells*", which are recurring design problems that affect the maintainability of test code. This thesis addresses a critical challenge: although there is an extensive catalogue of test smells proposed by academics (Garousi & Küçük, 2018), there is a lack of empirical evidence regarding their practical applicability. Evaluating the relevance of these test smells serves as the initial foundation of the thesis, which is crucial for the development of better tool support for developers. If test smells that offer little value to developers are increased, they may further hinder test suite maintainability. Hence, there is limited empirical evidence on whether developers recognize the maintainability issues posed by test smells, how they address them, and whether the presence of test smells correlates with defect densities. Despite this, researchers continue to develop tools for detecting and classifying test smells (Athanasiou, Nugroho, Visser, & Zaidman, 2014; Bavota, Qusef, Oliveto, De Lucia, & Binkley, 2015; Junior, Soares, Martins, & Machado, 2020a; Spadini, Palomba, Zaidman, Bruntink, & Bacchelli, 2018; Tufano et al., 2016). Hence, in the first aim of the thesis, we conducted (O1), which empirical re-evaluate and re-rank current perception of test smells by analyzing when and why developers remove test smells in software evolution. We also provide a regression-model to assess whether test smell may have an impact on defect-density, which is crucial for developing better test smell recommendation tools that

may have more impact of software quality.

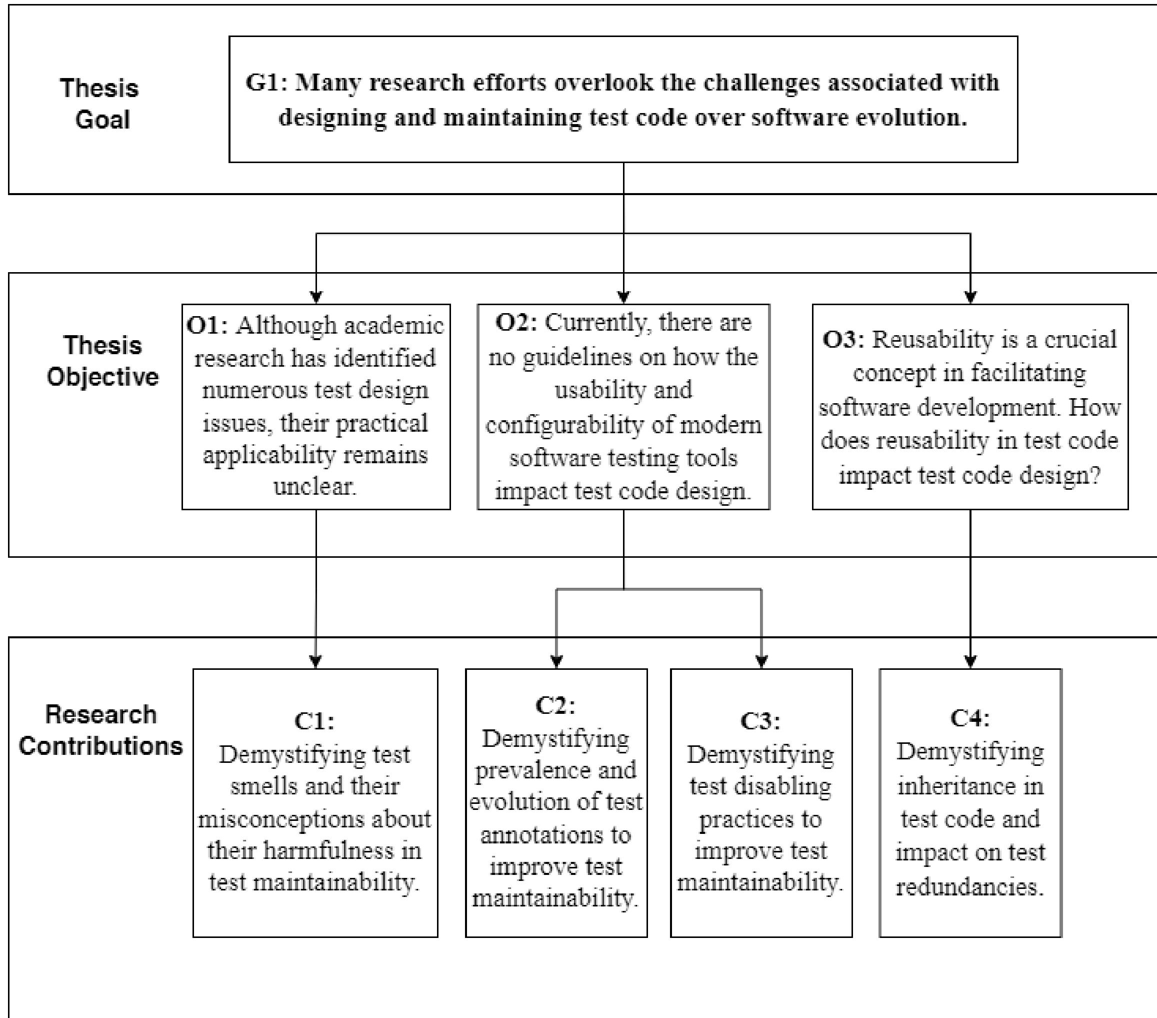

Figure 1.1: An overall view of this thesis

Objective 2 (O2): Currently, there is limited empirical evidence on how the usability and configurability of modern software testing tools impact test code design. ***Our objective is to provide insights on improving test code design based on modern software test automation frameworks.*** Current test smell catalogues are more than a decade old (Deursen, Moonen, Bergh, & Kok, 2001), and do not consider design issues that stems from the advancement in test automation tools. For example, testing framework like Junit, have changed the ways developers write and evolve test-suites, and hence they may be subjected to different kinds of maintainability issues that may hinder

test-suites code maintenance. Hence, in the second objective (O2) of the, we proposed two novel research directions to address this. In (C2), we conducted an empirical study on how developers leverage testing framework (e.g., Junit) involving annotations to improve test code maintenance (e.g., improve readability, find test flakiness, test performance), as prior studies (Zerouali & Mens, 2017) have found that JUnit4 is one of the most widely utilized testing frameworks to write annotation-based test code in Java-based systems.

The goal of effective software testing is to identify the presence of bugs through thorough program execution and verification of verdicts set by developers. The first step towards achieving fault-free software is bug identification. However, once a test fails due to verdict violation, fixing the bug can be non-trivial due to flakiness, difficulty in exact fault localization, and repair mechanisms. Due to time constraints for continuous delivery, these tests may sometimes be temporarily disabled. To address this, modern testing frameworks provide annotation APIs like @Ignore to help developers bypass failures. However, such practices result in technical debt in the test code. Hence, in (C3) we conducted an empirical study to demystify practices of disabling test code to understand to direct future research efforts: (a) understanding how bugs are fixed to improve automatic bug repair support, and (b) can help prevent future encounters with bugs that are hard to fix.

Objective 3 (O3): Understand reusability in test code design and its impact on the effectiveness of test code. While reusability is essential for facilitating software development, enabling developers to extend code, promote reuse, and enhance maintainability, test code reusability remains an underexplored area. Therefore, in (C4), our aim is to investigate reusability and extensibility in test code. Interestingly, we take a different view on reusability by examining paradigms like inheritance in object-oriented programming.

Our preliminary research shows that while inheritance may be controversial (Kainulainen, 2014; Stackoverflow, 2023), it is widely utilized to maintain test code. We conjecture that while inheritance improves ease of extensibility—for instance, inherited test cases are automatically executed to assess coverage—there are hidden costs associated with improper usage. For example, some inherited tests may be unnecessary yet executed, or tests inherited by multiple subclasses, where their behavior does not change, may lead to

redundancies. Hence, our aim is to demystify how inheritance is used to achieve reusability, often at the expense of introducing test case redundancies.

1.3 Thesis Contribution

Below, we delve into a detailed research contribution aimed at demystifying test code quality from the perspective of test maintainability.

1.3.1 Research Contribution for Objective 1

Based on our objective 1, we have contributed one research contribution.

Research Contribution 1 (C1): Demystifying test smells and their misconceptions about their harmfulness in test maintainability. Despite ongoing research into the discovery and classification of test smells (Athanasiou et al., 2014; Bavota et al., 2015; Junior et al., 2020a; Spadini et al., 2018; Tufano et al., 2016), there remains significant ambiguity and conflicting evidence regarding their impact on software maintainability. One research found that developers are aware of test smells and their potential consequences (Peruma et al., 2019a), while others found that developers do not believe the benefit of removing test smells (Junior et al., 2020a; Junior, Soares, Martins, & Machado, 2020b; Tufano et al., 2016). Therefore, our focus is to understand why developers remove test smells and their effects on software quality, particularly in terms of defect-proneness. We also build regression model to find the relationship between test smell addition/removal and software quality, in terms of defect-proneness.

1.3.2 Research Contribution for Objective 2

Based on our objective 2, we have contributed two research contribution (C2 and C3).

Research Contribution 2 (C2): Demystifying prevalence and evolution of test annotations to improve test maintainability. The introduction of annotations in Java 5 has driven annotation as a critical component of the many Java-based frameworks, influencing how developers to design and implement software. Even in software testing,

frameworks such as JUnit, TestNG, and Mockito have all adopted annotations as critical ingredients in test design and implementation. A prior study by Zerouali and Mens (2017) has found that JUnit4 is one of the most widely utilized testing frameworks for Java-based systems, and test annotations (e.g., @Test) are also one of the most widely used annotations in Java development. Despite the importance of test annotations, most prior research on test maintenance has only focused on general test design and test assertions (Athanasiou et al., 2014; Bavota, Qusef, Oliveto, De Lucia, & Binkley, 2012; Bavota et al., 2015; Garousi & Küçük, 2018; Junior et al., 2020a; Qusef, Elish, & Binkley, 2019) and has not considered the peculiarity of test annotations. Therefore, in this work, we aim to perform empirical study on how developers leverage test annotations to maintain test code quality (e.g., readability, test flakiness, test performance, obsolete test).

Research Contribution 3 (C3): Demystifying test disabling practices to improve test maintainability. While software testing plays an important role in identifying faults early, facilitating program repair, and potentially reducing future maintenance costs, repairing bug may not always be done immediately. Bugs may require multiple rounds of repairs and even remain unfixed due to the difficulty of bug-fixing tasks. To help test maintenance, along with code comments, the majority of testing frameworks (e.g., JUnit and TestNG) have also introduced annotations such as @Ignore to disable failing tests temporarily. Although disabling tests may help alleviate maintenance difficulties, they may also introduce technical debt. In this work we aim to perform empirical research on the prevalence, evolution, and maintenance of disabling tests in practice. We believe that studying why developers disable test code may help developers understand the source of potential technical debt that can direct future test maintainability practice.

1.3.3 Research Contribution for Objective 3

Based on our objective 3, we have contributed one research contribution (C4). **Research Contribution (C4): Demystifying inheritance in test code and impact on test redundancies.** Inheritance, a fundamental aspect of object-oriented design, has been leveraged to enhance code reuse and facilitate efficient software development. However,

alongside its benefits, inheritance can introduce tight coupling and complex relationships between classes, posing challenges for software maintenance. Although there are many studies on inheritance in source code, there is limited study on the test code counterpart. In this proposal, we aim to take the first step by studying inheritance in test code, with a focus on redundant test executions caused by inherited test cases. We proposed a hybrid approach based on static and dynamic test analysis to identify and locate test case redundancies introduced by inheritance that may increase test execution time but has no additional fault revealing capabilities. Most importantly, this work opens future research direction to further study why using inheritance in test code important, besides achieving reusability, and how can we better guide developers when using inheritance.

1.4 Thesis Overview

The outline of this thesis is as follows. Chapter 2 presents the background of this work and the basic concepts that shall be used throughout the thesis. In Chapter 3, we present the related works of our research on two dimensions: (1) empirical studies on re-evaluating and re-ranking current perception of test smells and their relationship with defect density, and (2) studies that propose more relevant test smells on use of modern testing framework. We also discuss the limitations of prior studies and the points that this thesis may complement previous research, and improve current state of test quality from perspective of test maintainability. Chapter 4 focuses on demystifying test smells and their misconceptions about their harmfulness in test maintainability. Chapter 5 focuses on demystifying prevalence and evolution of test annotations to improve test maintainability. Chapter 6 focuses on demystifying technical debt in test code by tracking test disabling practices. Chapter 7 focuses on demystifying test inheritance and their relationship with test redundancies. Chapter 8 concludes the thesis and discusses the future work.

Chapter 2

Background

In this chapter, we establish the background of this work and introduce the elementary concepts and the terminology that will be used throughout the thesis. Our work address four different problems within the broad area of software testing, specific to design issues and improve maintainability in the test code.

2.1 Context of the work

This thesis focuses on aiding developers to write maintainable test code by providing empirical assessments and methods for detecting design issues in the test code. Therefore, the work is situated on two research fields: empirical software engineering and software testing research. In the following section we introduce two fields:

2.1.1 Empirical Software Engineering

Empirical Software Engineering (ESE) is an area of research that emphasizes the use of empirical methods in the field of software engineering (*Empirical Software Engineering An International Journal Publishing model*, n.d.). This area focuses on improving software engineering processes by collecting, analyzing, and interpreting historical data mined from software repositories written by developers, using either quantitative or manual classification. Leveraging open-source historical data helps derive actionable insights to enhance

software engineering processes and quality.

2.1.2 Automated Software Testing

Software testing is a vital component of modern software development. It is a process that involves executing a software program and finding bugs so that the result will be defect-free software. With advancements in technology, there is increased research effort to automate the tedious nature of testing, considering various aspects of the testing process.

A plethora of research involves targeting three aspects of software testing research, as listed below:

(1) **Prioritizing the execution of fault-revealing tests** through regression test case prioritization (RTCP) (Beheshtian, Bavand, & Rigby, 2021; J. Chen et al., 2016; Jagannath, Luo, & Marinov, 2011), test case selection (RTCS) (Di Nardo, Alshahwan, Briand, & Labiche, 2015; Legunsen et al., 2016; S. Wang, Nam, & Tan, 2017), batch commit testing (Beheshtian et al., 2021), and test parallelization (Candido, Melo, & d'Amorim, 2017).

(2) **Localizing the buggy line** through spectrum-based techniques (utilizing coverage) (Abreu, Zoeteweij, & Van Gemund, 2006; Jones, Harrold, & Stasko, 2002; Keller et al., 2017) and information retrieval-based approaches (A. R. Chen, Chen, & Wang, 2021; T.-D. B. Le, Oentaryo, & Lo, 2015)

(3) Once a fault-revealing test fails and the presence of a bug has been ascertained, (3) automatically **repairing the buggy line** (X.-B. D. Le, Chu, Lo, Le Goues, & Visser, 2017; Mechtaev, Yi, & Roychoudhury, 2016; Ye, Martinez, Durieux, & Monperrus, 2021).

2.2 Test Quality

Despite existing efforts to improve test automation and effective bug resolution, there is considerably less focus on enhancing test quality. For instance, the efficacy of downstream

software testing techniques in ensuring software quality depends heavily on the quality of the tests initially written by developers. Without high-quality tests to verify the program's state during execution, downstream techniques for locating and fixing bugs may prove futile. To assess test quality, developers commonly use metrics such as coverage and mutation testing, which are de-facto industrial standards. In this thesis, we use the coverage metric but not the mutation score; therefore, we will omit discussing mutation testing in this section.

2.2.1 Code Coverage

Coverage quantifies the extent to which test code exercises the behavior of the source code, with higher code coverage increasing the likelihood of detecting bugs. Numerous coverage tools exist in the literature for Java-based systems, such as Cobertura (*cobertura*, 2024) and JaCoCo (Jacoco, 2023), which offer common coverage metrics like statement, branch, and path coverage. For example, as shown in Equation 1, statement coverage shows the percentage of statements in the software program that are executed by testing. If a software program has ten statements and six out of the ten statements have been executed (i.e., covered) by the test case, then the code coverage is 60%.

$$\text{Application Statement Coverage} = \frac{\text{Number of Statement executed by the Test Suites}}{\text{Total Number of Statements in the Application}} \tag{1}$$

2.2.2 Limitation of Common Test Quality Attribute

While both coverage may help address immediate concerns regarding the current quality of tests, they do not consider other test quality attributes that may impact test designs, such as maintainability, extensibility, and reusability of the test code. For example, prior work on automated test generation, such as Evosuite (Fraser & Arcuri, 2011) and Randoop (Pacheco & Ernst, 2007), has been shown to achieve high test quality by attaining both high coverage, and the ability to detect faults (Almasi, Hemmati, Fraser, Arcuri, & Benefelds, 2017; Fraser & Arcuri, 2015). Despite this, generated unit tests pose a significant maintenance burden,

making them harder to debug when included in a project (Ceccato, Marchetto, Mariani, Nguyen, & Tonella, 2015), as the automatically generated tests have poorer readability compared to their human-written counterparts.

2.3 Test Quality from Perspective of Test Maintainability, Extensibility and Reusability

To ensure the effectiveness of automated software testing, developers need to maintain a set of high-quality test cases together with its source code to continuously validate software quality. Unfortunately, similar to source code, test code may also contain defects and design issues that hinder the quality of the test code. Thus far, researchers and practitioners have started to notice recurring design problems in the test code (Spadini et al., 2018; Van Deursen, Moonen, Van Den Bergh, & Kok, 2001) and have coined the term *"test smell"*. Like code smells in source code, test smells indicate potential design problems that may negatively impact maintainability or readability of the test code.

2.3.1 Test Smells

Due to the increasing importance of design issues in test code, (Garousi & Küçük, 2018) conducted a large-scale systematic study to summarize a catalog of 196 test smell instances. However, most of the studied test smells are related to general code smells (e.g., long parameter list, god class, no comments, and bad naming), code smells specific to TCN language, and code smells from grey literature (i.e., blog posts) or difficult to generalize (e.g., complicated setup, long-running test, long test file). Different from their research, this thesis focuses on 18 other test smells because they are related to unit testing practices in Java (Peruma et al., 2020) and advocated in xUnit guidelines (Meszaros, 2007). For instance, traditional test smells have been proposed by Deursen et al. (2001), as listed below:

- **Assertion Roulette** (AR): One test case may contain several assertions with no explanation. AR increases difficulties in comprehension.

- **Eager Test** (EGT): A test case may exercise several methods of the object under test, which may increase the difficulty in test maintenance.
- **General Fixture** (GF): A test case's fixture is too general, and the test code only accesses a part of it. The test case may execute unnecessary code and increase runtime overhead.
- **Lazy Test**: Occurs when multiple test cases invoke the same method of the source code object, which may increase the difficulty in test maintenance.
- **Mystery Guest** (MG): Test code that uses external resources. Tests containing such a smell are difficult to comprehend and maintain, due to the lack of information to understand them.
- **Resource Optimism** (RO): A test case that makes optimistic assumptions about the state/existence of external resources, which may cause flaky test results.
- **Sensitive Equality** (SE): A test using the `toString` method for equality check in assert statements. The test case is sensitive to.

Motivated by the work by (Deursen et al., 2001), and recognizing the limited empirical foundation behind test smells in android systems, (Peruma et al., 2019a) proposed 11 new test smells, as listed below:

- **Conditional Test Logic** (CTL): There exist conditions in a test case that may alter the behavior of the test and its expected output.
- **Constructor Initialization** (CI): A test class may use a constructor instead of JUnit's `setUp()`. This may introduce side effects when the test class inherits another class, i.e., the parent class's constructor will still be invoked.
- **Empty Test** (ET): Occurs when test code has no executable statements.
- **Exception Catch/Throw** (ECT): Passing or failing of a test case depends on custom exception handling code or exception throwing, which may hide real problems and hamper debugging.

- **Print Statement** (PS): Print statements in unit tests are redundant as unit tests are executed as part of an automated script and do not affect the failing or passing of test cases. Furthermore, they can increase execution time if the developer calls a long-running method from within the print method (i.e., as a parameter).
- **Redundant Assertion** (RA): A test case may contain assertion statements that are either always true or always false.
- **Sleepy Test** (ST): Occurs when explicitly making a thread to sleep in test cases, which may cause flaky test results.
- **Duplicate Assert** (DA): Occurs when a test case tests the same condition multiple times, which may increase test overhead.
- **Unknown Test** (UT): A test method is written without an assertion statement.
- **IgnoredTest** (IT): A test case that is disabled using JUnit's @Ignore, which may increase compilation time, and increases code complexity and comprehension.
- **Magic Number Test** (MNT): A test method contains unexplained and undocumented numeric literals as parameters or identifiers, which increases maintenance difficulty.

As shown above, there are many catalogues of test smells proposed in the literature, yet it is unclear whether the presence of test smells poses real design issues in the test code. Understanding the impact of test smells through empirical assessment may provide insights for better tool support. There is a need to re-rank and re-evaluate the current perception of test smells, which we aim to address in Chapter 4 in this thesis.

2.4 Modern Java-Based Testing Framework

To aid the development of test suites and test automation, modern software development processes rely on testing frameworks, such as JUnit (Junit4, 2021) or TestNG (TestNG, 2023), in Java-based systems. These testing frameworks provide APIs that aid in writing

structural testing in three phases, known as the 3A: *Arrange*, *Act*, and *Assert* (Programming, 2011). *Arrange* initializes the small piece of application needed for testing, *Act* executes the application, and finally, *Assert* ensures that the results from the application equal the expected behavior. With the introduction of Java annotations, enforcing such testing patterns has become straightforward, such as adding the @Before annotation to enforce the initialization of dependencies (e.g., a database) that are required for setting up resources for testing, in the case of database systems. Below we describe the common annotation based testing API based on Junit4 to ease test suite development and automation:

2.4.1 Junit4 Test Automation

- **@Rule** (Field): @Rule provides a mechanism to enhance tests by running some code around a test case execution, which is similar to fixture and teardown.

- **@Parameterize** (Field/Method): Test cases annotated with @Parameterize can be invoked by using predefined inputs (i.e., parameterized test inputs) and expected output.

- **@Test** (Method): @Test indicates that the annotated test code should be executed as a test case. @Test takes optional parameters, such as Timeout to indicate that the test should finish within a given time, or exception to indicate that the test should throw an exception.

- **@Before/@After** (Method): @Before indicates that the annotated test code should be executed as a precondition before each test case (i.e., Database setup). Similarly, @After indicates the execution of the annotated test code as a postcondition after each test case.

- **@BeforeClass/@AfterClass** (Method): @BeforeClass and @AfterClass are similar to @Before and @After annotation types but indicate the annotated test code to only execute once (i.e., before or after the test class is invoked).

- **@Ignore** (Method/Class): @Ignore indicates that the annotated test case should not execute.
- **@Category** (Method/Class): @Category provides a mechanism to label and group tests, giving developers the option to include or exclude groups from execution.
- **@Test(timeout=X)** (Method/Class): A test will fail if its execution takes longer than the value X specified in timeout.

As modern software processes rely on testing frameworks to write better test suites, maintainability issues can arise from the usage or misusage of the API. We aim to elucidate test annotation maintenance in Chapter 5 of the thesis.

2.5 Fine-Grain Code Differencing for Mining Developers' Code Change History

Since software is never constant and must evolve to remain satisfactory (Mens et al., 2008), there is a need for a tool to detect code evolution history in order to identify design and code smells in software application.

2.5.1 Mining Code Change History

To address smells in software code, developers have introduced techniques like *refactoring*, which involves removing various design and code smells without affecting the behavior of the source code. Refactoring is important because its goal is to improve the maintainability of code with long-term sustainability in mind. For example, many researchers have empirically investigated the benefits of refactoring by studying how it improves reusability (Moser, Sillitti, Abrahamsson, & Succi, 2006), how the renaming of variable identifiers affects code readability (Silva, Tsantalis, & Valente, 2016), and how the introduction of Lambda expressions affects program comprehension (Lucas, Bonifácio, Canedo, Marcílio, & Lima, 2019; Shen et al., 2019). Due to the benefits of refactoring, researchers have introduced code differencing tools to track which refactoring operations were applied in version

history, helping developers better understand changes during code review (Alves, Song, & Kim, 2014; Ge, Sarkar, Witschey, & Murphy-Hill, 2017), resolve merge conflicts (Dig, Manzoor, Johnson, & Nguyen, 2008), and select better regression tests (K. Wang et al., 2018).

In this thesis, we utilize the most popular refactoring evolution miner tool called RMiner (Tsantalis, Ketkar, & Dig, 2020a) to mine test design and maintainability issues. We use RMiner because at its core, it is an AST differencing tool at the commit level that matches AST nodes of different types and supports semantic-aware changes. Moreover, it has the highest precision (99.6%) and recall (94%) among other refactoring mining and AST diff tools.

Chapter 3

Literature Review

In this thesis, we propose approaches to help developers improve test quality, with a focus on improving test design and test maintainability. Most prior studies in software testing primarily focus on downstream test automation technique, such as prioritizing fault revealing test (Di Nardo et al., 2015; Legunsen et al., 2016; S. Wang et al., 2017), automated fault localization (Abreu et al., 2006; Jones et al., 2002; Keller et al., 2017) and automated fault repair (X.-B. D. Le et al., 2017; Mechtaev et al., 2016; Ye et al., 2021). However, such approaches do not consider quality of developer written test from the perspective of test case design and maintainability. In addition, presence test design issues may pose a significant maintenance burden, i.e., harder to debug over-time, which may degrade software quality.

This thesis aims to improve test quality from the perspective of Software Engineering research by re-evaluating and re-ranking previously defined design issues (e.g., test smells) (Deursen et al., 2001; Peruma et al., 2019b). Furthermore, this thesis proposes a more relevant subset of test design issues that are applicable for modern software engineering process that relies on utilizing testing framework (e.g., JUnit for Java systems), to improve test maintenance and automation.

3.1 Paper Selection Process

For our literature review, we primarily focus on papers that analyze test design and maintainability issues in Java-based systems. Other literature reviews are included in dedicated chapters to emphasize more relevant subsets of research efforts associated with our research goals. Nevertheless, in this chapter, the chosen related works are mainly categorized into two aspects:

- **Empirical studies on re-evaluating practitioners' perceptions on test smells in real-world systems**: Test smells describe design issues that negatively impact maintainability in the test code. Since this concept is proposed from academic perspective, we report on studies that (1) aims to demystify whether practitioners care about these issue, and (2) whether test smells leads to higher defect density.
- **Research on improving test maintainability from testing framework and automation perspective:** Deviating from more traditional definition of test smells (Deursen et al., 2001), utilizing testing framework (e.g., Junit) is the best-practice by industrial standards to write better test code. In this section, we report on studies that focus on improving test design and maintainability from test framework perspective.

3.2 Empirical studies on practitioners perceptions' on test smells in real-world systems

In this section, we first discuss prior studies aimed at demystifying developers' perceptions of test smells and re-evaluating our current understanding of test smells. This knowledge is crucial for reassessing the prevailing notion that test smells are inherently harmful. In the latter part of this section, we discuss recent studies aimed at correlating test smells with defect density. These studies provide additional evidence that test smells negatively impact the maintainability of test code, potentially leading to more defects. This evidence further supports the notion that test smells may or may not be harmful in practice.

3.2.1 Perception and Maintenance of Test Smells

Akiyama (1971) discusses the importance of well-designed test code, arguing that well-crafted test cases are easier to comprehend and maintain. He proposes that refactoring production code differs from refactoring test code and suggests various types of test smell refactoring operations, such as removing dependencies and ensuring resource uniqueness. Building on this work, Deursen et al. (2001) introduces 11 catalogs of test smells, which represent patterns of poor design decisions associated with test code that impacts readability and maintainability.

Since the proposal of test smells, a study conducted by Tufano et al. (2016) surveyed developers' awareness of test smells. The results show that most developers do not recognize design problems in test code and do not perceive test smells as actual problems. Furthermore, to understand what kind of tool support is required, the study also conducts quantitative research to observe when test smells are introduced and fixed. Their results indicate that test smells have long survivability (i.e., 100 days), motivating the need for tool support to help address them. However, they do not consider the alternative hypothesis that developers may not care about test smells, hence there may be no need to address them.

Another survey by Junior et al. (2020a) suggests that developers' professional experience cannot be considered a root cause for the insertion of test smells in test code. However, it is worth noting that not all test smells necessarily result in problems, and perhaps specific types of test smells may require more attention due to other factors. Many prior studies have a preconceived notion that test smells are harmful, yet there is little empirical evidence to show that test smells are inherently harmful.

Peruma et al. (2019a) proposed a new catalog of test smells and a detection tool to address the lack of investigation of test smells in Android applications. They conclude that test smells are widely distributed and similar across both mobile and non-mobile application domains. Subsequently, they survey developers' awareness of these detected test smells, finding that developers are often aware of the negative consequences of test smells in the

software system. Their study contradicts prior findings by Tufano et al., which suggests that developers may struggle to justify addressing test smells as they may not directly impact quality attributes like defect density.

Spadini, Schvarcbacher, Oprescu, Bruntink, and Bacchelli (2020) use `tsDetector` to propose severity thresholds for ranking test smells. The new thresholds have been determined after investigating developers' perception of test smell severity. This represents a crucial first step towards re-evaluating the current perception of test smells, as not all test smells may be equally severe. However, there may be potential biases in their user study design, as not all types of test smells, such as assertion roulette, may indicate problems for test code (Panichella, Panichella, Fraser, Sawant, & Hellendoorn, 2022).

C. S. Yu, Treude, and Aniche (2019) is the first work to investigate the process involved in comprehending test code. They survey developers' time spent reading and extending test code at various test case design steps. While their research contributes to understanding factors influencing test code comprehension, gaining actionable insights for tool support proves challenging. The study lacks empirical evidence on the complex characteristics of test code evolution. Motivated by the limitations of current repair techniques to accurately design test cases, the study by Pinto, Sinha, and Orso (2012a) analyzes test code evolution in terms of modifications, additions, and deletions to elucidate the complicated evolution of test cases and suggest better repair techniques for the future. Similarly, we aim to fill the gap in empirical evidence regarding how developers may remove test smells in practice.

3.2.2 Using design issue as metric to understand software post-release defect

Software defect modeling has been proposed to ensure high quality by understanding the relationship between various software metrics (e.g., lines of code, McCabe's Cyclomatic complexity) and the software defects (Moser, Pedrycz, & Succi, 2008). Khomh, Di Penta, and Gueheneuc (2009) showed that the presence of code smells increases the code's change-proneness. Later on, they also showed that code components affected by code smells are more fault-prone than non-smelly components (Khomh, Penta, Guéhéneuc, & Antoniol,

2012). Their results were confirmed by Palomba et al. (2018), who found that code smells make classes more change and defect-prone. Spadini, Palomba, Zaidman, Bruntink, and Bacchelli (2018) showed that production code is more defect-prone when tested by smelly tests. However, Spinellis, Louridas, and Kechagia (2021) showed that code elements are durable, taking around 2 years to undergo modification, and this pattern shows no correlation with various quality attributes of the code, such as smells, based on their explainable regression analysis.

3.3 Research on improving test maintainability practices from perspective of test automation

Despite efforts to distill more relevant subsets of test smells to better guide developers in test maintainability, the current definitions of test smells are a decade old (Garousi & Küçük, 2018) and do not take into consideration design issues resulting from API misuse in modern testing frameworks. To address this gap, we first discuss prior studies aimed at improving test maintenance through the usage and adoption of modern testing tools.

From our empirical study on improving test case design through testing frameworks, we found that despite the advantages of using testing frameworks to enhance test case design and maintainability, API misuses may introduce test smells, leading to technical debt, specifically test debt, negatively impacting long-term maintainability (Cunningham, 1993). Therefore, in the latter section of the literature review, we delve into one additional relevant study: (2) an empirical investigation aimed at understanding test debt.

3.3.1 Improving test maintainability from framework usages

Researchers have been investigating how developers use various testing and mocking frameworks in open-source systems. Zerouali and Mens (2017) performed one of the first studies on the evolution of testing frameworks. Their study shows that JUnit is the most

prominent testing library, while many libraries, such as PowerMock, Mockito, and EasyMock, are often used simultaneously to complement each other. Researchers have also conducted quantitative studies to understand the adoption of mocking frameworks in mocking file dependencies (Marri, Tao Xie, Tillmann, de Halleux, & Schulte, 2009). More recently, Spadini, Aniche, Bruntink, and Bacchelli (2019) performed a comprehensive study on how and why developers use mocking in test code and how mocking evolves over time.

For example, Rocha and Valente (2011) mined 106 open-source Java systems to investigate annotation usage empirically. Similarly, Dyer, Rajan, Nguyen, and Nguyen (2014) analyzed 31K open-sourced projects to analyze how various Java language features are adopted by developers, including the adoption of Java annotations. Their study shows that Java annotations are very commonly adopted. With the increasing adoption of annotations in the development of programming languages, such as Java 5 and Python 3.5, annotations have become a critical component influencing how developers design and implement software. However, there is limited work in analyzing how annotations are used to improve test maintainability. Our aim is to derive valuable implications for researchers, developers, and testing framework designers to further expand and improve test annotation practices.

3.3.2 Empirical study on technical debt from the perspective of test code

Cunningham (1993) discussed the concept of technical debt, where a short-term rewards may induce higher maintenance costs in the long run. Building on this concept, Potdar and Shihab (2014) introduced the idea of self-admitted technical debt (SATD), where developers intentionally introduce commented-out code as a form of temporary fixes. Subsequently, Wehaibi, Shihab, and Guerrouj (2016) demonstrated that SATD introduces fewer future defects compared to non-SATD instances; however, implementing SATD changes is often more complex.

In the context of test debt, several studies have contributed relevant insights. Siebra et al. (2012) proposed inadequate testing as a form of test debt, while Brown et al. (2010) and Guo and Seaman (2011) discussed the concept of unfinished testing. Additionally, Wiklund, Eldh, Sundmark, and Lundqvist (2012) emphasized the importance of both automated and

manual test coverage to mitigate test debt. However, most literature on test debt focuses on the lack of testing infrastructure, but rarely delves into the reasons why test debt occurs. For example, temporary hot-fixes, such as disabling failing test cases due to the difficulty of bug fixing activities, may temporarily ease maintenance for developers but can introduce test debt due to a missing quality assurance activity.

Chapter 4

Demystifying test smells and their misconceptions about their harmfulness in test maintainability

In recent years, researchers and practitioners have been studying the impact of test smells on test maintenance. However, there is still limited empirical evidence on why developers remove test smells in software maintenance and the mechanism employed for addressing test smells. In this paper, we conduct an empirical study on 12 real-world open-source systems to study the evolution and maintenance of test smells, and how test smells are related to software quality. Our results show that: 1) Although the number of test smell instances increases, test smell density decreases as systems evolve. 2) However, our manual classification on those removed test smells reveals that most test smell removal (83%) is a by-product of feature maintenance activities. 45% of the removed test smells relocate to other test cases due to refactoring, while developers deliberately address the only 17% of the test smell instances, consisting of largely *Exception Catch/Throw* and *Sleepy Test*. 3) Our statistical model shows that test smell metrics can provide additional explanatory power on post-release defects over traditional baseline metrics (an average of 8.25% increase in AUC). However, most types of test smells have a minimal effect on post-release defects. Our

study provides insight into how developers resolve test smells and current test maintenance practices. Future studies on test smells may consider focusing on the specific types of test smells that may have a higher correlation with defect-proneness when helping developers with test code maintenance.

Earlier version of this chapter was published in Empirical Software Engineering Journal (EMSE 2021). 34 pages. (D. J. Kim, Chen, & Yang, 2021a)

4.1 Introduction

In modern software development, developers need to continuously implement changes to the software system to keep up with the consumers' ever-growing demands. As a software system evolves, tremendous collaborative effort takes place to deliver features and perform maintenance activities. Due to the importance of software quality, automated regression testing has played a pivotal role in software development. New test code is developed to test the newly-added code and is executed after code changes to ensure that the new changes do not introduce new defects (Ali et al., 2019).

To ensure the effectiveness of regression testing, developers need to maintain a set of high-quality test cases to continuously validate software quality. Unfortunately, similar to source code, test code may also contain defects and design issues that hinder the quality of the test code. For example, prior studies (Lam, Godefroid, Nath, Santhiar, & Thummalapenta, 2019a; Luo, Hariri, Eloussi, & Marinov, 2014a) have found that the results of some test cases may be unreliable (e.g., flaky tests) due to defects in test code. Thus far, researchers and practitioners have started to notice recurring design problems in the test code (Spadini et al., 2018; Van Deursen et al., 2001) and have coined the term *test smell*. Like code smells in source code, test smells indicate potential design problems in test code. Bavota et al. (2015) found that test smells are prevalent in software systems and may hinder test comprehension and maintenance.

Despite the findings achieved so far, there is limited yet conflicting empirical evidence on the awareness of test smells. One research found that developers are aware of test smells

and their potential consequences (Peruma et al., 2019a), while others found that developers do not believe the benefit of removing test smells (Junior et al., 2020a, 2020b; Tufano et al., 2016). Therefore, studying how and why developers address test smells will help expand future research on understanding what may prompt developers to maintain test code, and provide evidence on the most paid attention test smells.

In this paper, we conduct an empirical study on the maintenance of test smell in 12 large-scale open-source systems. We study a total of 18 different types of test smells that were defined and studied in prior research (Garousi & Küçük, 2018; Junior et al., 2020a, 2020b; Peruma et al., 2019a; Qusef et al., 2019; Spadini et al., 2018). In particular, we seek to answer the three following research questions:

RQ1: How do test smells evolve overtime? We conduct a quantitative analysis to study how tests smell evolve over a three year period (from 2016 to the beginning of 2019) in the studied systems. Although we find that the total number of test smell instances increases over time, the test smell density remains relatively stable in the 12 studied systems after normalizing by the total number of test code lines.

RQ2: What is the motivation behind removing test smells? We conduct a manual classification on a statistically significant sample of the commits that removed test smells. We find that developers directly address the test smells in only 17% of the sampled commits. In particular, developers are more likely to address two test smells: *Exception Catch/Throw* and *Sleepy Test*. However, in 83% of the studied commits, the test smells are removed due to the deletion of test code or are relocated to other test cases due to feature refactoring activities. In short, we find that developers often do not directly address the test smells when maintaining test code.

RQ3: What is the relationship between test smells and software quality? Similar to prior work (T. Chen, Shang, Nagappan, Hassan, & Thomas, 2017; de Pádua & Shang, 2018), we build a logistic regression model to study the relationship between test smell and software quality. We find that some test smells (e.g., *Conditional Test Logic*, *Exception Catch/Throw*, and *Mystery Guest*) have a positive correlation with a source code file's defect-proneness, after controlling for confounding factors like the traditional product, process and

coupling metrics. However, most types of test smells have minimal effect on the defect-proneness.

In summary, our findings show that, as a system evolves, developers may allocate resources on maintaining test code, but they may not be aware of the test smells. Moreover, some test smells have a minimal effect on defect-proneness, while only a few test smells have a positive impact on defect-proneness. Future studies on test smells may consider focusing on the types of test smells that may have a higher correlation with defect-proneness when helping developers with test code maintenance.

4.2 Background

4.2.1 Brief Overview of Test Smells

Section 2.3.1 shows the 18 different types of test smells that we include in our study. These test smells are studied in prior work (Bavota et al., 2012; Bavota et al., 2015; Garousi & Küçük, 2018; Junior et al., 2020a; Knuth, 1981; Peruma et al., 2019a). In particular, the current knowledge of test smells that we know from the literature was first proposed by Deursen et al. (2001), and these were expanded as a basis for further investigation in recent studies. For instance, some studies (Bavota et al., 2015; Bleser, Nucci, & Roover, 2019; Tufano et al., 2016) found a high diffusion of test smells in software systems, and such test smells may not be removed as systems evolve. Other studies investigated the impact of test smell on code comprehension by measuring the time taken for understanding the test code in the presence/absence of test smells (Bavota, Qusef, Oliveto, Lucia, & Binkley, 2012a). Moreover, Athanasiou et al. (2014) studied the impact of test smell on software quality (correlation with post-release defect) to fill the missing gap from numerous prior studies that only underlines its effects on software maintainability.

Numerous researchers also surveyed software engineers to understand their awareness, perception, or identification. For instance, a recent study by Peruma et al. (2019a) proposed a new set of test smells and investigated their diffusion and awareness. Their result suggests that developers are aware of test smells and their potential consequences. On the contrary,

others give evidence that developers do not believe software systems could genuinely benefit from addressing test smells (Junior et al., 2020a; Tufano et al., 2016). Nevertheless, there is a lack of empirical evidence on what types of test smell developers pay attention to the most and thereby maintain software evolution. Similarly, there is also missing evidence on the common reasons and mechanisms in which test smells are addressed. Hence, in this paper, we study how test smells evolve and how developers manage test smells during software maintenance. Moreover, we explore whether the existence and maintenance of test smells correlate with software quality. Our work uses the test smell detection tool implemented by Peruma et al. (2019a) which include the most comprehensive type of test smells up to date, encompassing both the test smells from the literature and their newly proposed test smells.

4.2.2 Identifying Test Smells

In this paper, we focus on studying the evolution and maintenance of test smells. To identify test smells, we adopt a test smell detection tool called *tsDetector* implemented by Peruma et al. (2019a) to analyze the studied systems. We choose *tsDetector* because it can detect a comprehensive list of test smells (i.e., 18 test smells in total, as described in Table 2.3.1) and has an average F-score of 96.5% (Peruma et al., 2019a). We focus on these 18 test smells because they are related to testing practices in Java (Peruma et al., 2020), advocated in xUnit guidelines (Meszaros, 2007), and extensively studied in prior researches in test code maintainability and developers' perception of test smells (Bavota et al., 2012; Junior et al., 2020a). Although Garousi and Küçük (2018) summarized a catalog of 198 test smells, many are general code smells specific to TCN language, come from grey literature (i.e., blog posts), and difficult to generalize (e.g., complicated setup, long-running test, and long test file). *tsDetector* uses JavaParser to detect test smell given the lists of the test files and the corresponding source code under test (i.e., *CUT*). The CUT files are required to detect specific types of test smells, such as *Eager Test* and *Lazy Test*, whose primary concerns are about testing multiple CUT files in one test case, which may negatively impact code comprehension.

To identify each test file's corresponding CUT files, we follow prior studies and utilize the naming convention (Chen, Thomas, Hemmati, Nagappan, & Hassan, 2017; Peruma et al., 2019a; Spadini et al., 2018; Tufano et al., 2016; Zaidman, Rompaey, Demeyer, & van Deursen, 2008). In particular, for each test file, we identify the corresponding CUT files by removing the prefix or the suffix of *"[Tt]est(s*)"* from the names of the test files. We also manually verify the build configuration files (e.g., Maven or Gradle build file) of the studied systems to use the default heuristic specified by Maven/Gradle plugin to identify test files. The default heuristic matches with the prefix that we use to determine the test files. The test smell detector *tsDetector* takes the lists of test files, and their associated CUT files and reports any occurrences of the 18 types of test smells.

Although *tsDetector* outputs test smells at a file-level, most reported test smells are at a line-level and method-level, which are aggregated per file. In the rest of our analysis, we study each test smell individually; therefore, at their respective line and method-level. Furthermore, we modify the *tsDetector* to output the raw count of test smells instead of the default boolean value. To encourage the replication of our results, we have made the dataset publicly available[1].

4.3 Studied Systems

We first introduce our studied systems. We then discuss the results of our research questions. For each research question, we discuss its motivation, the approach we use to address the question, and the results.

Case Study Systems. Table 4.1 shows an overview of the studied systems. We conduct our study on several versions of the 12 open-source Java systems. In particular, we conduct our research in all official releases from the beginning of 2016 to the beginning of 2019. We chose the studied systems based on the following selection criteria. First, we selected the top 1,000 Java projects on GitHub ordered by popularity (i.e., stargazer count). We also made sure that the repositories are not forks. Second, we discarded projects that are below

[1]https://github.com/SPEAR-SE/TestSmellEmpirical_Data

Table 4.1: An overview of the studied systems.

Systems	#Releases	LOC in Source Code (2016 - 2019)	LOC in Test Code (2016 - 2019)
Kafka	9	95K - 265K	18K - 101K
Groovy	9	338K - 393K	8K - 9K
Camel	8	586K - 1.0M	379K - 484K
Zookeeper	4	128K - 119K	25K - 36K
Cxf	9	696K - 753K	195K - 218K
Karaf	11	132K - 168K	14K - 17K
Flink	8	388K - 731K	100K - 234K
Accumulo	7	420K - 577K	49K - 47K
Hive	11	3.5M - 4.4M	162K - 221K
Bookkeeper	9	102K - 200K	32K - 85K
Wicket	8	264K - 257K	54K - 57K
Cassandra	6	315K - 184K	43K - 112K
Hadoop	3	637K - 1M	418K - 658K
Total	102	6.9M - 9.1M	1.1M - 1.6M

the 90th percentile in terms of size (i.e., lines of code), repository popularity (i.e., stars), and the number of commits. We also remove systems that do not use issue report systems. In the end, we are left with these 12 systems.

As shown in Table 4.2, there are 998 active contributors in total (ranges from 15 to over 200 contributors) in the studied systems with a wide range of experiences (i.e., in terms of number of commits). In some systems, such as Kafka and Flink, the contributors' median number of commits is relatively high (i.e., 306 and 278 commits, respectively), which shows that many contributors are actively contributing to the systems. The studied systems are widely used by practitioners, used in many commercial settings, and are large in scale, with the number of lines of code (LOC) in source code ranges from 6.9M to 9.1M, and the LOC in test code ranges from 1.1M to 1.6M. Moreover, the studied systems maintain a set of comprehensive test cases and adopt the continuous integration practice by running the test cases daily basis (Apache, 2020). The studied systems also cover different domains, from big data processing and data warehousing solutions to distributed databases and programming languages.

Table 4.2: An overview of the developer experience and the number of contributors in the studied systems.

	# Contributed Commits						
Systems	Min	Q1	Median	Q3	Max	Mean #Commits	Contributor
Accumulo	1	5.75	27.00	486.25	2528	7.16	26
Bookkeeper	18	40.75	87.00	217.25	2545	4.37	35
camel	1	6.00	31.00	217.25	24339	6.61	204
Cassandra	2	23.50	101.50	189.50	1320	3.20	102
Cxf	1	3.00	10.00	82.75	8767	4.88	47
Flink	1	98.50	278.00	791.50	3456	3.25	175
Groovy	2	24.00	68.00	530.25	909	1.34	15
Hive	5	28.50	122.50	325.00	3041	2.80	112
Kafka	20	53.00	306.00	607.00	918	3.24	188
Karaf	1	1.00	2.00	61.00	949	1.91	38
Wicket	1	7.00	34.00	320.00	4219	1.39	20
Zookeeper	3	12.50	47.50	92.75	671	2.03	36
Hadoop	1	54.74	225.50	591	2368	2.47	373

4.4 Results

4.4.1 RQ1: How do test smells evolve overtime?

Prior studies (Bavota et al., 2015; Tufano et al., 2016) reveal that test smells are prevalent in software systems, and their presence hinders the comprehension and maintenance of test code. In light of these findings, there is limited empirical evidence of how the pervasiveness of test smell changes over time and its relation to software maintenance. In this RQ, we quantitatively investigate the evolution of test smells.

Approach: To study the evolution of test smells, we follow the approach described in Section 4.2 to detect test smells in each studied software version. In particular, we apply a test smell detection tool called *tsDetector* (Peruma et al., 2019a) to analyze the studied systems based on six-month windows from 2016 to 2019. In total, we obtain seven snapshots per studied system. We consider as six-month window because studying the evolution of test code on a commit by commit basis is expensive and dilute the modifications of test smells. Moreover, since the studied systems have different sizes and test smells may co-evolve with the amount of added test code and the raw number of the test smell instances, we also report test smell density. We calculate test smell density by dividing the number of test smell instances by the total number of code lines. We use code lines as our normalization metrics because many test smells are detected at the line level. We also normalized using

other metrics such as the number of methods in a file, and we found a similar trend.

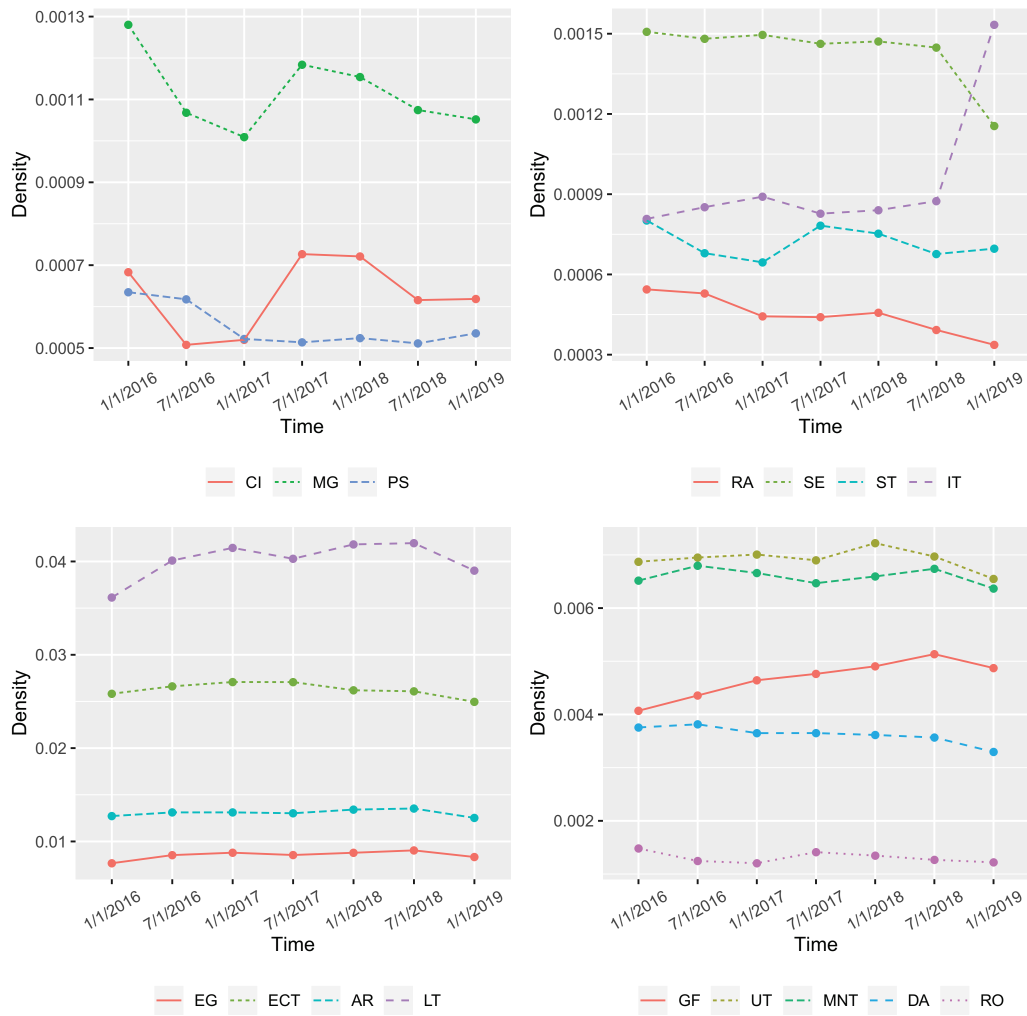

Figure 4.1: Time series plots that show the evolution of the test smell density (normalized average) of the studied systems. The test smell densities are calculated based on seven snapshots that are taken every six months between 2016 and 2019.

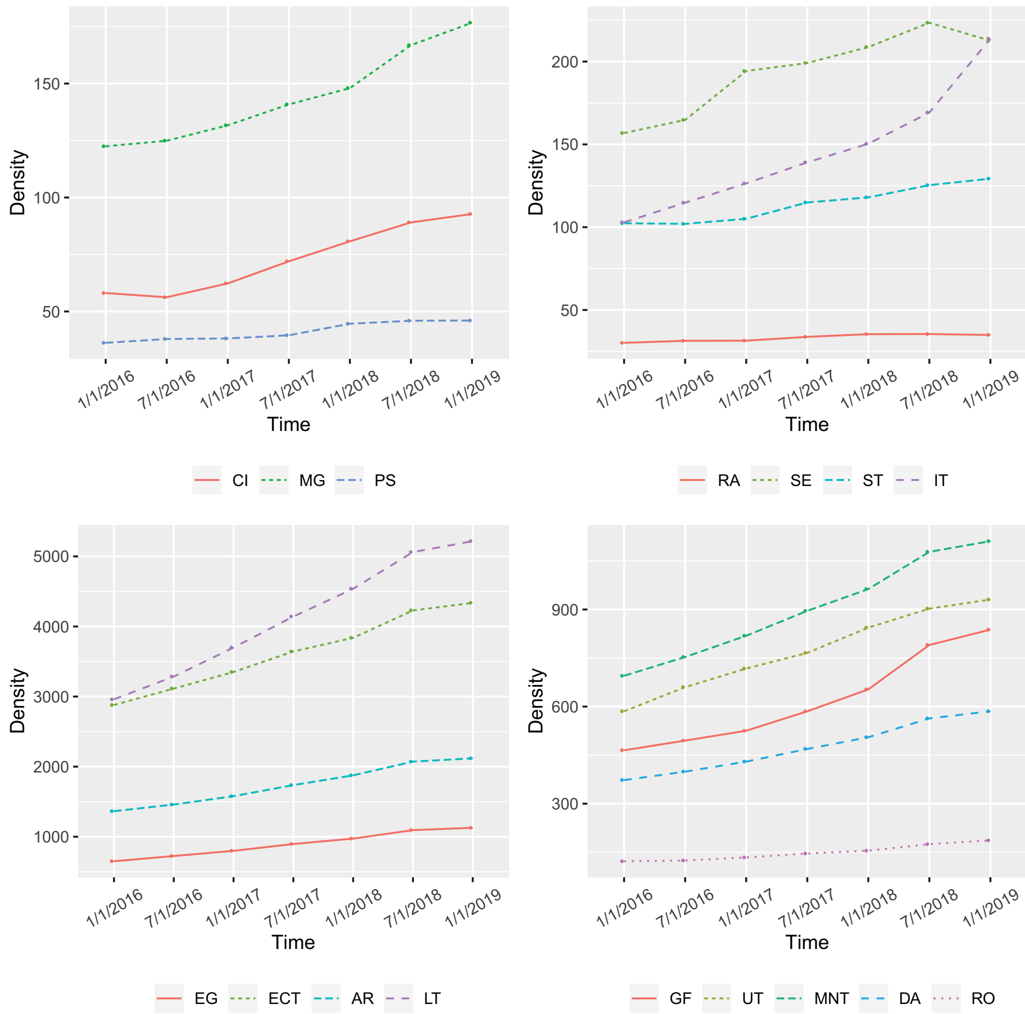

Figure 4.2: Time series plots that show the evolution of the averaged raw test smell of the studied systems. The test smells are aggregated from on seven snapshots taken every six months between 2016 and 2019.

Results: Figure 4.1 shows the time series plots of the average test smell density in the studied systems from 2016 to 2019. We present the studied systems with a similar scale of test smell densities in one plot for the ease of visualization. Average test smell density is the normalized test smell instances that is averaged over all studied systems. We averaged the test smell densities to show a generalized trend amongst the studied systems. We find that most test smell densities also stay relatively stable. However, ignored test smells (IT) increased far greater over-time compared to other test smells. Figure 4.2 shows the evolution of the raw counts of test smell instances averaged over all of the studied systems.

In general, we observe that all of the averaged raw test smell metrics increase over-time, but normalized test smell densities remain stable.

Table 4.3: The comparison of the test smell density (number of test smell instances per 1000 lines of test code) for each type of test smell in the studied systems from 2016 and 2019.

Test Smell	Accumulo			Bookkeeper			Camel			Cassandra			Cxf		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	15.63	14.87	-5 % ↗	6.17	11.53	61 % ↘	12.58	13.61	8 % ↘	8.18	9.78	18 % ↘	14.67	14.69	-
CTL	5.39	5.45	1 % ↘	7.97	8.07	1 % ↘	3.47	3.78	9 % ↘	7.96	7.09	-12 % ↗	4.16	4.20	1 % ↘
CI	0.41	0.27	-40 % ↗	2.34	1.53	-42 % ↗	0.21	0.32	38 % ↘	0.06	0.17	101 % ↘	0.79	0.74	-6 % ↗
ET	0.05	0.02	-79 % ↗	0.00	0.00	-	0.19	0.14	-28 % ↗	0.00	0.00	- % ↘	0.06	0.05	-12 % ↗
ECT	19.67	15.88	-21 % ↗	19.46	24.50	23 % ↘	33.55	34.63	3 % ↘	21.85	20.75	-5 % ↗	29.11	28.33	-3 % ↗
GF	6.35	5.57	-13 % ↗	5.79	6.54	12 % ↘	2.05	2.37	14 % ↘	1.37	2.41	55 % ↘	4.26	3.93	-8 % ↗
MG	0.82	0.55	-40 % ↗	1.87	1.54	-19 % ↗	1.09	0.91	-17 % ↗	1.17	1.19	1 % ↘	1.40	1.27	-10 % ↗
PS	0.12	0.06	-63 % ↗	0.06	0.12	60 % ↘	0.08	0.06	-18 % ↗	0.35	0.27	-27 % ↗	0.08	0.10	24 % ↘
RA	0.92	0.34	-93 % ↗	0.06	0.32	134 % ↘	0.10	0.06	-52 % ↗	0.25	0.30	16 % ↘	0.12	0.12	-
SE	2.15	1.56	-32 % ↗	0.03	0.22	150 % ↘	0.85	0.77	-10 % ↗	0.28	0.51	59 % ↘	2.16	2.00	-7 % ↗
ST	0.56	0.55	-2 % ↗	3.01	1.47	-68 % ↗	1.32	0.94	-33 % ↗	0.87	0.76	-14 % ↗	0.47	0.53	12 % ↘
EG	14.11	15.22	8 % ↘	1.96	9.17	129 % ↘	3.11	3.51	12 % ↘	5.12	5.95	15 % ↘	6.71	6.49	-3 % ↗
LT	84.91	81.97	-4 % ↗	6.01	36.97	144 % ↘	11.92	15.00	23 % ↘	29.79	32.10	7 % ↘	24.74	24.40	-1 % ↗
DA	4.37	4.29	-2 % ↗	3.73	4.21	12 % ↘	2.04	2.47	19 % ↘	2.98	3.30	10 % ↘	3.03	3.01	-1 % ↗
UT	5.90	5.40	-9 % ↗	0.85	4.24	133 % ↘	3.46	4.28	21 % ↘	7.59	6.64	-13 % ↗	7.63	6.67	-13 % ↗
IT	0.46	0.61	28 % ↘	0.70	0.90	25 % ↘	0.86	1.10	25 % ↘	2.08	2.10	1 % ↘	0.94	0.78	-20 % ↗
RO	0.99	0.69	-35 % ↗	2.18	1.85	-17 % ↗	0.85	0.91	7 % ↘	1.22	1.24	2 % ↘	1.12	1.03	-9 % ↗
MNT	6.23	6.96	11 % ↘	3.04	6.62	74 % ↘	6.15	6.65	8 % ↘	5.45	5.75	5 % ↘	6.35	7.22	13 % ↘

	Flink			Groovy			Hive			Kafka			Karaf		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	12.24	12.92	5 % ↘	19.52	22.44	14 % ↘	13.17	11.46	-14 % ↗	16.67	16.83	1 % ↘	10.96	11.84	8 % ↘
CTL	5.16	4.21	-20 % ↗	7.84	6.25	-23 % ↗	5.06	4.36	-15 % ↗	4.55	3.96	-14 % ↗	2.53	2.28	-10 % ↗
CI	0.73	0.70	-4 % ↗	0.77	0.74	-4 % ↗	0.64	0.76	17 % ↘	0.17	0.16	-9 % ↗	0.49	0.37	-28 % ↗
ET	0.04	0.10	83 % ↘	0.51	0.32	-47 % ↗	0.03	0.03	-12 % ↗	0.06	0.02	-98 % ↗	0.07	0.05	-28 % ↗
ECT	16.63	19.28	15 % ↘	64.35	58.01	-10 % ↗	23.08	25.94	12 % ↘	13.70	12.97	-5 % ↗	37.18	36.69	-1 % ↗
GF	0.57	2.85	133 % ↘	5.01	6.99	33 % ↘	5.68	6.08	7 % ↘	3.90	11.30	97 % ↘	2.18	2.87	27 % ↘
MG	0.80	0.87	8 % ↘	1.03	0.85	-19 % ↗	0.43	0.50	16 % ↘	2.04	0.47	-125 % ↗	1.97	1.81	-9 % ↗
PS	0.10	0.03	-120 % ↗	3.21	1.91	-51 % ↗	0.50	0.37	-30 % ↗	0.00	0.07	200 % ↘	2.88	3.35	15 % ↘
RA	0.04	0.03	-31 % ↗	3.08	1.59	-64 % ↗	0.93	0.75	-21 % ↗	0.06	0.13	76 % ↘	0.07	0.05	-28 % ↗
SE	0.26	0.45	52 % ↘	3.73	3.18	-16 % ↗	2.89	1.82	-46 % ↗	0.17	0.57	106 % ↘	0.56	0.90	46 % ↘
ST	0.56	0.39	-36 % ↗	0.00	0.00	-	0.15	0.25	49 % ↘	0.12	0.11	-6 % ↗	0.35	1.33	116 % ↘
EG	7.21	9.16	24 % ↘	9.38	15.24	48 % ↘	7.30	6.23	-16 % ↗	17.25	17.39	1 % ↘	5.83	7.49	25 % ↘
LT	42.04	45.36	8 % ↘	43.42	75.80	54 % ↘	34.12	33.66	-1 % ↗	78.91	80.87	2 % ↘	21.44	28.04	27 % ↗
DA	3.74	3.47	-8 % ↗	5.39	4.23	-24 % ↗	4.49	4.28	-5 % ↗	4.31	3.30	-27 % ↗	1.83	1.65	-10 % ↗
UT	3.31	5.32	46 % ↘	15.29	13.66	-11 % ↗	6.99	8.03	14 % ↘	4.43	5.12	14 % ↘	17.36	15.03	-14 % ↘
IT	0.54	1.23	78 % ↘	0.64	0.53	-19 % ↗	0.60	0.89	38 % ↘	0.29	0.38	26 % ↘	1.69	2.60	43 % ↘
RO	0.89	0.93	4 % ↘	2.57	1.80	-35 % ↗	0.53	0.56	5 % ↘	2.10	0.59	-113 % ↗	2.32	2.23	-4 % ↗
MNT	4.90	5.44	10 % ↘	11.43	10.69	-7 % ↗	8.70	7.45	-15 % ↗	11.07	8.06	-32 % ↗	5.69	6.27	10 % ↘

	Wicket			Zookeeper			Hadoop		
	2016	2019	%	2016	2019	%	2016	2019	%
AR	17.36	3.24	-137 % ↗	8.50	9.69	13 % ↘	9.70	9.92	2 % ↘
CTL	2.09	0.51	-121 % ↗	8.30	8.17	-2 % ↗	5.61	5.12	-9 % ↗
CI	1.29	1.20	-8 % ↗	0.56	0.66	17 % ↘	0.41	0.41	1 % ↘
ET	0.11	0.00	-200 % ↗	0.04	0.03	-36 % ↗	0.04	0.04	-11 % ↗
ECT	13.56	2.43	-139 % ↗	22.19	23.76	7 % ↘	21.41	21.31	-
GF	5.31	0.12	-191 % ↗	4.27	5.85	31 % ↘	6.15	6.47	5 % ↘
MG	0.20	0.11	-63 % ↗	2.75	2.46	-11 % ↗	1.07	1.16	7 % ↘
PS	0.13	0.02	-152 % ↗	0.16	0.14	-15 % ↗	0.58	0.48	-19 % ↗
RA	0.59	0.09	-148 % ↗	0.68	0.47	-36 % ↗	0.18	0.14	-22 % ↗
SE	4.52	0.62	-152 % ↗	0.84	0.72	-15 % ↗	1.15	1.70	39 % ↘
ST	0.00	0.04	200 % ↘	2.00	1.77	-12 % ↗	1.03	0.92	-11 % ↗
EG	12.86	2.27	-140 % ↗	3.87	5.08	27 % ↘	4.93	5.12	4 % ↘
LT	57.07	10.53	-138 % ↗	13.09	19.60	40 % ↘	22.38	22.99	3 % ↘
DA	5.07	0.81	-145 % ↗	4.23	4.20	-1 % ↗	3.58	3.64	1 % ↘
UT	5.41	0.65	-157 % ↗	6.62	5.74	-14 % ↗	4.48	4.36	-3 % ↗
IT	0.17	7.32	191 % ↘	0.44	0.47	7 % ↘	1.09	1.03	-5 % ↗
RO	0.33	0.07	-130 % ↗	2.91	2.68	-8 % ↗	1.23	1.28	4 % ↘
MNT	5.52	1.02	-138 % ↗	4.55	4.89	7 % ↘	5.62	5.78	3 % ↘

Table 4.4: The comparison of the prevalence of test smells (i.e., the raw number of test smell instances) for the studied systems from 2016 to 2019.

Test Smell	Accumulo			Bookkeeper			Camel			Cassandra			Cxf		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	647	707	9 % ↗	195	978	134 % ↗	4740	6573	32 % ↗	585	1317	77 % ↗	2845	3219	12 % ↗
CTL	223	259	15 % ↗	252	685	92 % ↗	1309	1828	33 % ↗	569	955	51 % ↗	807	921	13 % ↗
CI	17	13	-27 % ↘	74	130	55 % ↗	81	153	62 % ↗	4	23	141 % ↗	153	163	6 % ↗
ET	2	1	-67 % ↘	0	0	-	71	69	-3 % ↘	0	0	-	11	11	-
ECT	814	755	-8 % ↘	615	2079	109 % ↗	12641	16727	28 % ↗	1562	2795	57 % ↗	5644	6209	10 % ↗
GF	263	265	1 % ↗	183	555	101 % ↗	773	1144	39 % ↗	98	324	107 % ↗	826	861	4 % ↗
MG	34	26	-27 % ↘	59	131	76 % ↗	409	442	8 % ↗	84	160	62 % ↗	271	278	3 % ↗
PS	5	3	-50 % ↘	2	10	133 % ↗	29	31	7 % ↗	25	36	36 % ↗	16	23	36 % ↗
RA	38	16	-81 % ↘	2	27	172 % ↗	37	28	-28 % ↘	18	40	76 % ↗	23	26	12 % ↗
SE	89	74	-18 % ↘	1	19	180 % ↗	321	372	15 % ↗	20	69	110 % ↗	418	439	5 % ↗
ST	23	26	12 % ↗	95	125	27 % ↗	496	456	-8 % ↘	62	102	49 % ↗	91	116	24 % ↗
EG	584	724	21 % ↗	62	778	170 % ↗	1171	1698	37 % ↗	366	802	75 % ↗	1302	1422	9 % ↗
LT	3514	3898	10 % ↗	190	3137	177 % ↗	4490	7245	47 % ↗	2130	4324	68 % ↗	4797	5346	11 % ↗
DA	181	204	12 % ↗	118	357	101 % ↗	770	1194	43 % ↗	213	444	70 % ↗	588	659	11 % ↗
UT	244	257	5 % ↗	27	360	172 % ↗	1305	2067	45 % ↗	543	894	49 % ↗	1479	1462	-1 % ↘
IT	19	29	42 % ↗	22	76	110 % ↗	324	533	49 % ↗	149	283	62 % ↗	183	170	-7 % ↘
RO	41	33	-22 % ↘	69	157	78 % ↗	320	442	32 % ↗	87	167	63 % ↗	218	225	3 % ↗
MNT	258	331	25 % ↗	96	562	142 % ↗	2318	3211	32 % ↗	390	774	66 % ↗	1232	1583	25 % ↗

Test Smell	Flink			Groovy			Hive			Kafka			Karaf		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	1204	3031	86 % ↗	152	212	33 % ↗	1721	2533	38 % ↗	286	1692	142 % ↗	156	223	35 % ↗
CTL	508	988	64 % ↗	61	59	-3 % ↘	662	964	37 % ↗	78	398	134 % ↗	36	43	18 % ↗
CI	72	165	78 % ↗	6	7	15 % ↗	84	168	67 % ↗	3	16	137 % ↗	7	7	-
ET	4	23	141 % ↗	4	3	-29 % ↘	4	6	40 % ↗	1	2	67 % ↗	1	1	-
ECT	1636	4521	94 % ↗	501	548	9 % ↗	3017	5734	62 % ↗	235	1304	139 % ↗	529	691	27 % ↗
GF	56	669	169 % ↗	39	66	51 % ↗	743	1344	58 % ↗	67	1136	178 % ↗	31	54	54 % ↗
MG	79	205	89 % ↗	8	8	-	56	111	66 % ↗	35	47	29 % ↗	28	34	19 % ↗
PS	10	6	-50 % ↘	25	18	-33 % ↘	65	81	22 % ↗	0	7	200 % ↗	41	63	42 % ↗
RA	4	7	55 % ↗	24	15	-46 % ↘	121	166	31 % ↗	1	13	171 % ↗	1	1	-
SE	26	106	121 % ↗	29	30	3 % ↗	378	402	6 % ↗	3	57	180 % ↗	8	17	72 % ↗
ST	55	91	49 % ↗	0	0	-	20	56	95 % ↗	2	11	138 % ↗	5	25	133 % ↗
EG	709	2149	101 % ↗	73	144	65 % ↗	954	1378	36 % ↗	296	1749	142 % ↗	83	141	52 % ↗
LT	4135	10637	88 % ↗	338	716	72 % ↗	4460	7442	50 % ↗	1354	8131	143 % ↗	305	528	54 % ↗
DA	368	814	75 % ↗	42	40	-5 % ↘	587	946	47 % ↗	74	332	127 % ↗	26	31	18 % ↗
UT	326	1247	117 % ↗	119	129	8 % ↗	914	1775	64 % ↗	76	515	149 % ↗	247	283	14 % ↗
IT	53	288	138 % ↗	5	5	-	79	196	85 % ↗	5	38	153 % ↗	24	49	68 % ↗
RO	88	219	85 % ↗	20	17	-16 % ↘	69	123	56 % ↗	36	59	48 % ↗	33	42	24 % ↗
MNT	482	1275	90 % ↗	89	101	13 % ↗	1137	1647	37 % ↗	190	810	124 % ↗	81	118	37 % ↗

	Wicket			Zookeeper			Hadoop		
	2016	2019	%	2016	2019	%	2016	2019	%
AR	941	184	-135 % ↘	213	351	49 % ↗	4057	6527	47 % ↗
CTL	113	29	-118 % ↘	208	296	35 % ↗	2344	3367	36 % ↗
CI	70	68	-3 % ↘	14	24	53 % ↗	170	269	45 % ↗
ET	6	0	-200 % ↘	1	1	-	17	24	34 % ↗
ECT	735	138	-137 % ↘	556	861	43 % ↗	8953	14014	44 % ↗
GF	288	7	-191 % ↘	107	212	66 % ↗	2571	4254	49 % ↗
MG	11	6	-59 % ↘	69	89	25 % ↗	449	760	51 % ↗
PS	7	1	-150 % ↘	4	5	22 % ↗	242	314	26 % ↗
RA	32	5	-146 % ↘	17	17	-	74	93	23 % ↗
SE	245	35	-150 % ↘	21	26	21 % ↗	480	1118	80 % ↗
ST	0	2	200 % ↗	50	64	25 % ↗	432	607	34 % ↗
EG	697	129	-138 % ↘	97	184	62 % ↗	2062	3365	48 % ↗
LT	3093	598	-135 % ↘	328	710	74 % ↗	9358	15116	47 % ↗
DA	275	46	-143 % ↘	106	152	36 % ↗	1499	2393	46 % ↗
UT	293	37	-155 % ↘	166	208	22 % ↗	1872	2865	42 % ↗
IT	9	416	192 % ↗	11	17	43 % ↗	455	680	40 % ↗
RO	18	4	-127 % ↘	73	97	28 % ↗	516	841	48 % ↗
MNT	299	58	-135 % ↘	114	177	43 % ↗	2351	3798	47 % ↗

We further investigate the change in the magnitude of test smell instances (i.e., raw counts) and test smell density between the two snapshots taken in 2016 and 2019. Table 4.3 shows the change in the magnitude of the test smell density, and similarly, Table 4.4 shows the change in the number of test smell instances (raw counts). As shown in Table 4.3, the test smell density decreases for most types of test smells. On the contrary, we find that the test smell instances' raw counts increase for most types of test smells (Table 4.4). Namely, 190 (81%) out of 234 (i.e., 18 test smell types times 12 studied systems) of the test smell types (across all studied systems) have an increase in the number detected test smell instances, which indicates that test smells are prevalent in the software and gradually grow over time. However, after normalized by the LOC of the test code, 121 out of 234 (51%) of the test smell types (across all studied systems) have a decreased test smell density. The findings may indicate that while the number of added test smell instances are higher as the systems evolve, test smell addition may be slower than that of test code addition. In other words, either developer may introduce fewer test smell instances when adding new test code or actively maintain test code, which results in the removal of test smell instances. We further study the reason for test smell removal in RQ2.

Discussion: Since developers with higher experience may fix more test smells, we further study the correlation between developers' experience and test smell removal/addition. Following a prior study by Rahman and Devanbu (2011), we use the number of previous commits as a proxy for developers' experience. From the beginning of 2016 to the beginning of 2019, we mined all the commits that modified the test file and calculated the test smell removal/addition for each unique contributor. Then we study the relationship between test smell removal/addition and developers' experience (i.e., in terms of the number of prior commits) by employing Spearman's rank correlation coefficient. We choose Spearman's rank correlation since it is a non-parametric correlation test that does not assume the underlying data distribution. We found a positive correlation (i.e., 0.53) between test smell addition and developers' experience and found a negative correlation (i.e., -0.57) between test smell removal and developers' experience. **The correlation analysis suggests a non-negligible correlation that experienced developers are more likely to add**

more test smells and remove fewer test smells. One potential explanation for our result is that even highly experienced developers often do not refactor test smells due to lack of awareness or benefits, which aligns with a prior survey (Peruma et al., 2019a). Another reason may be that most experienced developers work on the most often exercised and most complex part of the system (Zeller, 2009). To corroborate our result, we study the relationship between the size of code changes (i.e., lines of code and deleted) and developers' experience. We employ a quantile-based correlation, where we split developers into four different quantiles based on their experiences and studied their correlation with test smell changes. Our result shows an increase in the correlation between experience and test smell addition (i.e., 0.037, 0.194, 0.262, 0.215), and decrease in correlation between experience and test smell removal (i.e., -0.090, -0.227, -0.248, -0.157). Therefore, we observe that the high expertise team may not necessarily remove more test smells because they may be responsible for larger and more complex changes.

> Although the number of test smells increases as the systems evolve, after normalizing against Test_{LOC}, the test smell density generally remains stable. We also find that some types of test smell, such as *Eager Test*, *Ignored Test*, *Unknown Test*, *Lazy Test*, and *Sleepy Test*, have one of the largest increase in terms of test smell density in most studied systems. In contrast, *Exception Catch/Throw*, *Redundant Assertion*, and *Print Statement* have the largest decrease in terms of test smell density in most studied systems.

4.4.2 RQ2: What is the motivation behind removing test smells?

A recent study (Garousi & Küçük, 2018) claims that developers perceive test smell as harmful in software systems. In contrast, other studies reveal that developers are unaware of test smells and do not acknowledge the benefits of refactoring them (Junior et al., 2020a; Tufano et al., 2016). Nevertheless, there is limited empirical support on test smell removals, whether a test smell vanishes as a side-effect of code evolution or is a refactoring target. Such evidence is necessary to reveal the current perception developers may have on test

smells. Hence, we perform a manual classification on the test smell removing commits to identify the reasons that prompt developers to fix the test code with test smells and the mechanisms employed to address test smells.

Approach. We conduct a manual classification on commits removing test smells. We leverage Git to extract all the commits except for the merge commits between 2016 and 2019. We only keep the commits that include test file modifications and discard the other commits from further analysis. A commit modifies a test file if the involved files have the extension *".java"* and have a prefix or a suffix of *"[Tt]est(s*)"*. For each of the commits, we run *tsDetector* on the two versions of the software (i.e., every two consecutive commits) and calculate the change in the number of test smell instances.

To understand why developers remove test smells, we take an indirect approach by looking at a combination of bug reports, commit messages, test code, and the commit history of relevant test code. Using the artifacts above, we answer three types of questions: 1) What kind of maintenance activity initially prompted developers to address test smells (i.e., the main purpose of the commit)? 2) How is the test smell addressed (e.g., is it refactored or by-product of other maintenance activities)? 3) When developers remove test smells, do their discussions in the software artefacts align with the definitions of test smells, providing evidence of their awareness? We used the three preliminary inquiries as a proxy to gain insight into developers' awareness of the most common reason for removing test smell.

In the analysis, we take a statistically significant sample of the commits removing test smells. In particular, we apply stratified random sampling on these commits with a 95% confidence level and a 5% confidence interval. We use a 95% confidence level and a 5% confidence interval because this provides a high probability that the true population parameter lies within 5% of our sample estimate. We adopted stratified sampling to sample the test smells in each studied system independently, which can be advantageous to reduce sampling error when a subpopulation within the overall population varies (Zhao, Liang, & Dang, 2019). In total, we manually analyzed 304 commits from a total of 1,452 commits

(achieving a 95% confidence level with a 5% confidence interval). The first two authors examine the sample independently. Any disagreement is discussed until reaching a consensus. We used Cohen's Kappa inter-rater agreement to measure the degree of agreement between the two authors (Cohen, 1960). Cohen's kappa considers a scenario where the agreement between two authors is purely by chance. We achieve Cohen's Kappa coefficient of 0.97, indicating almost a perfect agreement level Cohen (1960).

To assist our manual classification, we leveraged a tool called Refactoring Aware Commit Review (Tsantalis, Mansouri, Eshkevari, Mazinanian, & Dig, 2018a), which is a code diff visualization tool for showing the refactoring activities applied between two commits. In the studied samples, 241 commits (80%) includes issue ID from the Jira bug report, and 25 commits (8%) use GitHub's pull request/issue tracking. The remaining 34 commits (11%) only contained commit messages. Some commit messages contained sufficient information to understand the reason behind test code changes. Such commit messages may include keywords like *"Refactor"* or *"Fix test speed."*

Result. Table 4.5 shows a two-dimensional summary illustrating the association between the maintenance activities that initially prompted developers to maintain test code (horizontal dimension) and the type of specific test code changes (vertical dimension) that developers applied when removing test smells. We also found 12 incorrectly detected test smell instances by *tsDetector*, i.e., a 4% false-positive rate, and excluded them in Table 4.5.

For the maintenance activities (horizontal), we uncover five categories that prompted developers to maintain test code. Four of the five categories are: refactoring test code (65 commits), feature improvement (87 commits), bug fixing (76 commits), and adding new functionality (55 commits). The remaining nine commits (i.e., the fifth category - "others") consist of the ones that we cannot identify clear motives due to insufficient documentation (e.g., low-quality commit messages and bug reports). As an example, in CXF (*bcb6385a*), the developer addresses a test smell (i.e., *Ignored Test*) by completing the test implementation. However, the test case was ignored when it was first introduced to the codebase years ago and did not reference any bug report. Thus, it is difficult to label the correct

Table 4.5: A summary of our manual classification on the commits that removes test smell, i.e., 304 sampled commits minus 12 commits that are incorrectly flagged by the test smell detection tool. Our analysis focuses on the context of each commit and how the test smell is addressed. In particular, we show the association between test code changes and the corresponding maintenance activities that developers apply.

	Maintenance Activities					
Code change	Refactoring Test Code	Feature Improvement	Bug Fixing	Feature Addition	Others	Total #
TEST SMELL AWARE REFACTORING						
Exception Catch/Throw	9	4	3	5	-	21
Sleepy Test	9	4	1	1	-	15
Unknown Test	-	3	1	1	2	7
Assertion Roulette	1	-	1	-	-	2
Sensitive Equality	-	-	2	-	-	2
Magic Number	1	-	-	1	-	2
Conditional Test Logic	-	1	-	-	-	1
Total	20	12	8	8	2	50
TEST SMELL UNAWARE REFACTORING						
Persistence	10	22	12	13	1	58
By-Product Removal	20	18	22	10	5	75
Total	30	40	34	23	6	133
OTHER CODE CHANGES						
Test Code Deletion	13	33	30	20	1	97
Add Comment/@Ignore	2	1	1	3	-	7
Revert a Commit	-	1	3	1	-	5
Total	15	35	34	24	1	109
#Total	65	87	76	55	9	292

maintenance motives.

We classify the type of test code changes (vertical) into three categories: *Test Smell Aware Refactoring* (50 commits), *Test Smell Unaware Refactoring* (133 commits), and *Other Code Changes* (109 commits). We classify a commit in the category of *Test Smell Aware Refactoring* if developers directly addressed the test smell. We classify a commit as a *Test Smell Unaware Refactoring* if the test smells are removed as a side-effect of other activities. *Test Smell Unaware Refactoring* consists of two subcategories: *Test Smell Persistence* and *By-product Removal*. The *Test Smell Persistence* (58 commits) shows instances of applying standard code refactoring, such as extracting common test code, where test smells are transiently relocated to another test class. The *By-product Removal* (75 commits) represents the cases when the test smell is removed due to refactoring and maintenance of other tasks (e.g., removing duplicate source code). We group the remaining commits that remove test smells but are not test smell specific refactoring into the *"Other Code Changes"* (109 commits). These commits made changes such as deleting test code, disabling tests (commenting or ignoring code), and reverting a commit.

In the following subsections, we discuss our manual classification results of the three high-level test smell removal categories.

Test Smell Aware Refactoring

Although less frequent, we find that developers directly refactor specific test smells in 50 out of 292 (17%) studied commits. As shown in Table 4.5, these commits are related to removing *Exception Catch/Throw* (21 commits), *Sleepy Test* (15 commits), *Unknown Test* (7 commits), *Assertion Roulette* (2 commits), *Sensitive Equality* (2 commits), *Magic Number* (2 commits), and *Conditional Test Logic* (2 commits). By looking into the vertical dimension (i.e., the context of the commits), we find that *Test Smell Aware Refactoring* happen more frequently during test refactoring commits (20/50), but developers also remove test smells during other maintenance activities (e.g., feature improvement and bug fixing).

However, not all test smell are removed by developers (i.e., developers may remove the test code completely). Therefore, we also report the proportion of test smell *"fixing"*

commits (i.e., removed by developers) over the number of test smell removing commits in our stratified samples (i.e., all commits that remove the specific test smell), which shows the true proportion of the fixed test smells. In this case, even though *Exception Catch/Throw* has the highest number of test smell removing commits, only 31% were removed. *Sleepy Test* has the highest number of removed (60%) compared to other test smells. 21% of *Unknown Test*, 12.5% of *Sensitive Equality*, 5.1% of *Magic Number*, 2.9% of *Assertion Roulette*, and 1.9% of *Conditional Test Logic* are removed by developers. Our findings show that, in the studied systems, developers are more likely to fix *Sleepy Test* and *Exception Catch/Throw* due to the existence of test smell instead of other maintenance activities. Below, we discuss how developers address different types of test smells.

Refactor Sleepy Test. 15 out of the 292 (5%) commits are related to refactoring the *Sleepy Test* test smell. This represents cases where developers removed 60% of the removed *Sleepy Test*. This test smell occurs when developers explicitly cause a thread to sleep, leading to unexpected results as the processing time for a task can differ on different devices (Meszaros, 2007). We find that developers often remove *Sleepy Test* due to unexpected test behavior and increased test time. For example, in Kafka (*7b7c4a7*), a developer mentions that:

"The timeouts are often large (e.g., 10 seconds) and still occasionally they trigger prematurely. They need to be replaced by waitUntilTrue and some logic that checks when processing in streams is complete".

In another example, a developer in Camel (*722e590c*) mentions that *"[u]se awaitility for testing where we otherwise use thread sleep which can be speeded up."*. We find developers often apply two approaches to address *Sleepy Test*. One is to use `waitFor()` condition in the Java *Awaitility* library and the other is to refactor the test smell using Java's *Future* library. These two approaches allow the test case to run asynchronously without blocking.

As presented in Table 4.3 (RQ1), while *Sleepy Test* accounts for one of the most prevalent test smells, we find in our manual classification that developers also allocate some efforts to refactor such smells. The awareness for *Sleepy Test* may be a result of the increase in attention for the unreliability in test code qualities (e.g., flaky tests) (Eck, Palomba, Castelluccio, & Bacchelli, 2019; Lam, Godefroid, et al., 2019a; Shi, Bell, & Marinov, 2019).

Moreover, we find that developers may also be concerned with an increased test execution time caused by calling `thread.sleep()`. Future research is needed to understand further developers' awareness and opinion on the consequences of *Sleepy Test*.

Refactor Exception Catch/Throw. We find that 21 out of the 292 (7.2%) commits are related to refactoring the *Exception Catch/Throw* test smell. This represents cases where developers removed 31% of the removed *Exception Catch/Throw* (other instances were removed as the by-product of other maintenance activities). As discussed in a previous paper (Peruma et al., 2019a), this test smell occurs when the passing or failing of the test is dependent on custom exception handling code or exception throwing instead of using JUnit's expected attribute. In this category, developers refactored the test smell in 10 commits, and the remaining were refactored during other maintenance activities: feature improvement (3 commits), bug fixing (3 commits), and feature addition (5 commits). As shown in Listing 4.1, developers remove the code logic in the catch block that determines the passing and failing of the test case. The developers mentioned in the bug report that using fail in the catch block is a bad programming style and masks the details of the stack trace[2]. After removing the test smell, a new test smell (i.e., unknown test) is introduced. We have also seen other similar cases in our study, where a new test smell is introduced after developers resolved the current test smell. **In short, our manual classification finds that developers are more likely to refactor the *Exception Catch/Throw* test smell during various maintenance tasks**. We also find that these test smells removal is often not associated with test failures but to improve future maintainability.

Unknown Test. We find that developers refactor *Unknown Test* in 7 out of 292 (<3%) commits. This represents cases where developers removed 21% of the removed *Unknown Test*. This test smell occurs when test cases do not contain any logic or assertions statements. Thus, it is challenging to comprehend what the role of the test case is. As an example, in Kafka (*7d6ca52a*), the test class called *JmxReporterTest.java* is added three years ago. However, three years after its creation, the developers noticed missing test code while working on other tasks and immediately addressed it. Similarly, as mentioned

[2] https://github.com/apache/flink/pull/4446

Listing 4.1: Developers removed the dependency of test outcome on exception handling code (Flink - *e83217bd*).

```
1  @Test
2  public void testZeroSizeHeapSegment() {
3      - try {
4              MemorySegment segment = new HeapMemorySegment(new byte[0]);
5              testZeroSizeBuffer(segment);
6              testSegmentWithSizeLargerZero(segment);
7      - }
8      - catch (Exception e) {
9      -   e.printStackTrace();
10     -  fail(e.getMessage());
11     - }
12 }
```

in CXF (*bcb6385*), developers completed the missing test implementation two years ago. **Thus, our finding suggests that there might be other instances of the *Unknown Test*, where developers may only notice them while maintaining other tasks**. One potential reason for adding *Unknown Test* code may be that in feature additions, the *Unknown Test* gets added to prepare for the future implementation. This is illustrated in CXF (*2705f4d*), *[CXF-7525] Completing the system test*, where developers initially only provided empty test cases when implementing the feature and later complete the test case. While adding *Unknown Tests* may serve as code documentation to describe what test cases should be implemented in the future; developers may forget to complete the test case and become technical debt (Cunningham, 1993; Pham & Yang, 2020a).

Refactor Sensitive Equality. We find that developers refactor the *Sensitive Equality* test smell in 2 out of 292 (<1%) commits. This represents cases where developers deliberately fixed 12.5% of the removed *Sensitive Equality*. This test smell occurs when the test method verifies objects by invoking the toString() method. The potential consequence of the test smell is that the change in the implementation of toString() might result in test failure (Meszaros, 2007). We find a similar discussion in Flink (*390d3613*), "*rerollercoasting through abstraction layer; we don't really know what the implementation is by calling tostring?*".

While there were a total of 16 samples of the removed *Sensitive Equality*, only 2 out of

16 commits (12%) reflected an awareness of the test smell (in other cases, developers delete the entire test method or move the whole test code to another test case). It may be because the use of the default toString() is intuitive from both its purpose and naming convention, and developers may not have an immediate incentive to address the test smell until the test fails.

Refactor Magic Number. We find that developers refactor *Magic Number* in 2 out of 292 (<1%) commits. This represents cases where developers removed 5.1% of the removed *Magic number*. This test smell occurs when assert statements in a test method contain numeric literals (i.e., Magic Numbers) as parameters. *Magic Number* does not indicate the meaning/purpose of the number. Hence, they should be replaced with constants or variables, thereby providing a descriptive name for the input (Meszaros, 2007). As an example in Kafka (*7ebc5da6*), the test smell was refactored after a feature addition, which involved explicitly replacing the *Magic Number* with a variable with a more meaningful variable name to improve code comprehension.

Figure 4.3: Assertion refactoring, removing duplicate assertion and Assertion Roulette test smells. Located in the commit 3b42fb5 from Apache Karaf.

```
1    @Test
2    public void testLoad() {
3            .....
4             dependency = conditional.getCondition().get(0);
5             String actual = "req:osgi.ee;filter:=\"(
                  (osgi.ee=JavaSE)(!(version>=1.7)))";
6          - assertNotNull(dependency);
7          - assertEquals(actual, dependency);
8          + assertThat(dependency, contains(actual));
9        }
10   }
```

Refactor Assertion Roulette. We find that 2 out of 292 (<1%) commits are from refactoring *Assertion Roulette* (AR). This represents cases where developers removed 2.9% of the removed *Assertion Roulette* in the manual samples. This test smell occurs when the test method has several assertion statements making it challenging to determine which assertion had failed (Meszaros, 2007). Although prior work (Deursen et al., 2001; Meszaros, 2007)

Figure 4.4: Conditional logic refactoring, removing complex conditional logic using assertions. Located in the commit 7ebc5da6 from Apache Kafka.

```
1    @Test
2    public void checkTypeInformation() {
3     .....
4     - if(tupleType.isTupleType()) {
5     -    if(!((TupleTypeInfo<?>)tupleType).equals(testTupleType)) {
6     -   fail("Tuple type information was not set correctly!");
7     -    }
8     - } else {
9     -  fail("Type information was not set to tuple type information!");
10    - }
11    + assertThat(tupleType.isTupleType(), is(true));
12    + assertThat(tupleType, is(equalTo(expectedType)));
13    .....
14   }
```

proposes using an assertion explanation to refactor the test smell, we find that developers may also remove the test smell using another assertion statement. Listing 4.3 shows an example where assertThat is used to remove both *Assertion Roulette* and duplicate assertion test smell. In this case, developers attempt to mitigate the test code's verboseness by refactoring with assertions, which helps to remove the test smells.

Conditional Test Logic. We find that 1 out of 292 (<1%) commits removed *Conditional Test Logic*. The finding shows that developers removed 1.9% of the removed *Unknown Test*. This test smell occurs when the test case's success or failure depends on the assertion method within the control flow blocks and thus is not predictable (Meszaros, 2007). A prior survey (Garousi & Küçük, 2018) noted that developers prefer to consider it is smelly or not on a "case-by-case basis". Our study also finds that developers typically do not refactor *Conditional Test Logic*. For the only case that we found, developers refactored the test smell when there are nested conditional statements. In Kafka (*7ebc5da6*), shown in the code snippet in Listing 4.4, the developer simplifies the test smell's verbosity with assertion statements. Namely, assertThat() & is() is used to improve the readability of the test logic.

Although less frequent, we find that developers removed specific test smells in 50 out of 292 (17%) studied commits. In particular, *Exception Catch/Throw* and *Sleepy Test* are the two most commonly refactored test smells. Based on the discussion in bug reports and commit messages, we find that developers are often aware of the sub-optimal practice of using Thread.sleep and spent efforts on improving the design of exception handling mechanisms. Even though the number is less, we also find some refactoring of other test smells, such as the *Unknown Test*, *Magic Number*, *Sensitive Equality*, *Conditional Test Logic*, and *Assertion Roulette*.

Test Smell Unaware Refactoring

Most test smells (133/292, 45%) are not removed by developers but are either relocated or deleted as consequences of other refactoring activities. We classify such commits as *Test Smell Unaware Refactoring* since developers were unaware of the test smells and removed them as a by-product of other maintenance tasks, such as refactoring, feature improvement, feature addition, or bug fixing. In our manual classification, we find that *Test Smell Unaware Refactoring* may affect the test smells in two ways: 1) The test smells are relocated to another codebase, i.e., test smell persistence. 2) The test smells are removed unintentionally due to cascading results of other source code refactoring activities. Below we discuss the two categories in detail.

Test Smell Persistence. We find that in 58 out of 292 (20%) commits, the test smells were relocated to other locations. In this case, test code may undergo various refactoring activities such as introducing inheritance (19 commits), extracting method (25 commits), extracting class (10 commits), replacing method with the existing helper (2 commits), and moving method/class (2 commits). For example, in CXF (*31a4a55*), developers applied two refactorings (i.e., extract superclass and pull up method) to the test case *JCacheOAuthDataProviderTest* to extract common test code. However, four existing test smells (i.e., *Assertion Roulette*, *Conditional Test Logic*, *Duplicate Assertion*, and *Magic Number*) in the test code are relocated to the new test case as a result of the refactoring. **Namely, developers did not remove the test smells during test code refactoring. In some**

cases, relocation may magnify test smell's effect. For example, in Hive (*14e92703*), while the developer fixes a bug associated with test failure, the developer extracts a reusable method that explicitly causes a thread to sleep and relocates code to a method in a test utility file. The method was then used in three other test cases, thus magnifying the test smell's impact.

By-Product Removal. We find that in 75 out of 292 commits (25%), the test smells are removed by developers as a cascading effect of source code changes. Such maintenance activities include feature improvement, adding features, and bug fixing (i.e., the horizontal view in Table 4.5). For example, a commit in Accumulo (*9dadca0f*) implements a new feature and refactors the source code using a builder design pattern. As a result of the source code changes, one test smell instance of *Eager Test* (i.e., calling multiple source code methods in a test case) is removed since the test case now calls the builder method instead of invoking four distinct methods.

In summary, we find that developers may refactor test code while performing other maintenance activities. Developers may refactor for future maintainability or as a necessary precursor for change in feature requirement. For example, to support new features in the source code, developers may refactor test code to accommodate common logic in test code and apply code reuse. However, in most cases, test smells were unintentionally removed by test code relocation or diffusion as a side-effect of these maintainability tasks. Our findings show that developers are unaware of test smells and may not actively remove test smells as systems evolve.

> We find that developers may not directly remove test smells when refactoring test code. Many manually studied test smells (133/292, 45%) are relocated or removed unintentionally by developers.

Other Maintenance Activities

We find that test code may be deleted as a system evolves. The majority of test smell

removals are related to test code deletion. 109 out of 292 (37%) commits belong to the category *Other Code Changes*. We classify a commit into this category when the removed test smell results from deleting test code, disabling test (ignoring/commenting out), or reverting a commit.

Test Code Deletion. In our study, we find that for a non-trivial number of commits (97/292 commits, 33%), the test smells are removed because the test code is deleted. Developers may delete a test case when it becomes redundant or obsolete. For example, in Flink (*e671f34*), while porting source code to another file, developers discuss the removal of test cases since another already has similar test cases. Developers sometimes also delete test cases when they become obsolete or hard to maintain as the system evolves. For example, in Accumulo (*c265ea5b*), the test code becomes irrelevant since the corresponding features under test are unstable and removed. Therefore, the test smells in the test code are also removed.

Add Comment/@Ignore. 7 out of 292 (2%) commits are related to commenting out or ignoring the test code. This category represents removing test smells as a result of temporarily disabling the test code. In general, we find that developers may comment out the entire test case to bypass test failure. For example, in Kafka (*ca1f18e*), developers commented out the test code to temporarily make the test pass since the test would only work after developers migrate to Java 9. As another example, in Camel (*9ad68066*), the test case is ignored due to test failure caused by a recent upgrade to jetty 9.3. Although the test smell is removed due to commenting or ignoring the test code, the test smell is not addressed. Lastly, we also see cases where the commented out test case was only brought back a few months later. Future studies should also investigate if such commented out test cases are re-enabled or become a technical debt in the system (Pham & Yang, 2020a).

Revert a Commit. 5 out of 292 (<2%) commits are related to reverting the test code. This category represents removing test smells as a result of temporarily reverting software to the previous versions. For example, in Wicket (*266c90037*), the system was reverted due to a defect caused by adding new features. Thus, the newly introduced test smell was also reverted.

We find that as the system evolves, test code and its associated test smells may be deleted due to the obsoleteness and maintenance difficulty of the source/test code. Developers may also temporarily comment out test cases to bypass test failures caused by recent code changes. However, we see instances where developers only bring the commented out test code back after several months or years.

Summary & Implication. Our manual classification shows that, in most cases, developers may not be aware of the test smells. We find that 82.9% of the studied test smells are removed, relocated, or disabled (e.g., commented out) as a by-product of other maintenance activities. During these refactoring activities, developers may relocate the test smell to another test case, and the test smell remains unchanged. In some cases, as discussed, the impact of test smell may become larger, as the test code that contains test smells is extracted to become a utility method. Nevertheless, we still find that developers removed test smells in 16% of the studied commits. In particular, we find that developers are more likely to remove *Exception Catch/Throw* and *Sleepy Test*. **Our finding suggests that, although developers may refactor test code, they often do not remove test smells**. In the next RQ, we further investigate the relationship between test smells and software quality.

4.4.3 RQ3: What is the relationship between test smells and software quality?

Although researchers have made a necessary step towards understanding the maintainability aspects of test smells (Bavota et al., 2015; Bleser et al., 2019; Junior et al., 2020a; Peruma et al., 2019a), it is still not clear whether removing test smells has an effect on the quality of software. In the previous RQ, we find that in addition to deliberately resolving test smells, test smells are commonly removed as by-products of other maintenance activities, such as deleting the test code entirely. Regardless of how the test smells are removed or relocated (RQ2), the removed test smells are no longer impacting the source

code files. However, it remains unknown whether test smells have any relationship with the code quality. Hence, in this RQ, we aim to understand further the relationship between test smells and software quality, particularly the post-release defect. Our finding may help identify the types of test smells that correlate with a post-release defect and inspire future research that helps developers efficiently address more test smells.

Approach. Our goal is not to predict defects but to study the additive effect of test smell metrics on post-release defects over controlled metrics using logistic regression models.[3] Logistic regression models are commonly used in prior research to study the effect of various software metrics on post-release defect (Bird, Nagappan, Murphy, Gall, & Devanbu, 2011; Chen, Thomas, Nagappan, & Hassan, 2012; de Pádua & Shang, 2018). Below, we define the metrics that we use and the model building process.

Studied Metrics & Data Collections

- ***Post-Release Defects.*** The post-release defect is our response metric in the regression model. The post-release defect is defined as the defects reported within a fixed time frame after a certain version of a software is released (Moser et al., 2008; Piotrowski & Madeyski, 2020). For each source code file, we label it as defect prone if the file is modified at least once in bug fixing commits within six months after the release of the software system (Zimmermann, Premraj, & Zeller, 2007). Developers in the studied systems are required to enter the issue ID in commit messages. Thus, we first query the issue tracker to obtain a list of bug reporting issues within six-month of each release date. We then find all the bug-fixing commits based on whether the commit messages contain one of the obtained issue IDs. At the end of the step, we obtain the list of source code files that contain post-release defects (e.g., TRUE or FALSE). Finally, if a test file tests a source code file that contains a post-release defect, we label the test file as defect-prone.

- ***Traditional Product and Process Metric.*** Similar to prior studies, we control

[3] Logistic Regression from Lrm R package.

for traditional product and process metrics in our regression model. Previous studies found that traditional product metrics (e.g., lines of code) and process metrics (e.g., code churn and pre-release defect) are good explainers for post-release defects (Moser et al., 2008; Nagappan & Ball, 2005; Nagappan, Ball, & Zeller, 2006) and are commonly used as baseline metrics (Bird et al., 2011; Chen et al., 2012; D'Ambros, Lanza, & Robbes, 2010). We collect these metrics at the test-file level and use them as a baseline to build a BASE model. We later add test smell metrics to the BASE model and study whether the test smell metrics may further explain a source code file's defect-proneness. Although our metrics may not represent all of the metrics, they are shown to have a high correlation with other complexity metrics and used for benchmarking in prior proposals of new metrics (Biyani & Santhanam, 1998; T. Chen et al., 2017; D'Ambros et al., 2010). For the traditional product metric, we used *CLOC* (AlDanial, 2019) to extract the LOC metric in the test file. For traditional process metrics, we use commands "git follows" and "git diff" to extract three different code churn metrics: file churn, code churn, and deleted lines of code. File churn is the number of commits that modified the file. Code churn is the total number of code lines, such as code deletion, addition, and modification. Finally, code deletion is the total lines of code deleted. For the other process metric, namely the pre-release defect metric, we follow a similar approach to extracting post-release defects using a six-month time window before one software release.

- ***Coupling Metric.*** In addition to the traditional product and process metrics, we also add two coupling metrics (namely COUPLING) as our controlled metrics to the BASE model to reduce the effects of confounding variables. We measure two coupling metrics, *ts_coupling* (i.e., a test case to source code) and *tt_coupling* (i.e., test cases to test cases), in test cases, which are used in prior studies to assess the quality of test code design (Child, Rosner, & Counsell, 2019). We exclude the dependencies with external frameworks or libraries when calculating the coupling metrics.

- ***Test Smell Product and Process Metrics.*** We consider both test smell product

and process metrics. Test smell product metrics (TEST_PRODUCT) are the number of detected test smells present in the system's current release. Test smell process metrics (TEST_PROCESS) are the number of test smells added and removed six months before releasing the system. Note that we extract the two metrics for each type of test smells (18 types in total). Calculating test smell process metrics can be challenging due to file deletion and rename. To address these challenges, we use the "git follow" to keep track of package change and file renaming. Since test smells are detected at different granularities, such as line, method, and class level, we aggregated the test smells at the file-level.

Model Construction

We use a logistic regression model to model post-release defect because it is easier to interpret and is widely used in prior studies (T. Chen et al., 2017; Harrell Jr, 2015; Kuhn & Johnson, 2013; Nagappan & Ball, 2005). Logistic regression can better isolate (with a predominantly additive effect) the effects of the test smell metrics on explaining post-release defect over the BASE model (i.e., an improvement on the model fitness) (Harrell Jr, 2015). In particular, we build an initial model using the baseline metrics (i.e., traditional process and product metrics and coupling metrics). Then, we build a series of new models to add TEST_PRODUCT and TEST_PROCESS over the BASE model. By studying the explanatory power of a series of models and their additive effects of test smell metrics, we explore whether test smell contributes to a better explanation of post-release defect. We build three models for each studied system:

- **BASE (LOC+CHURN+PRE+COUPLING)**: The baseline model uses the traditional product, process, and coupling metrics.
- **BASE+TEST_PRODUCT**: We add TEST_PRODUCT to the BASE model and measure the improvement in the explanatory power over the BASE model.
- **BASE+TEST_PRODUCT+TEST_PROCESS**: The third model measures the

combined effect of TEST_PRODUCT and TEST_PROCESS metrics over the BASE model.

For each model that we construct, we first apply data transformation to reduce the data skewness. We follow prior studies using log-transformation on the metrics to normalize the data (Chen et al., 2012; de Pádua & Shang, 2018). Second, we remove the metrics with a zero variance because these metrics do not contribute to the model (i.e., the values are constant). Third, we apply redundancy analysis to drop predictors that can be predicted based on a model composed of all other predictors with an adjusted R^2 of higher than 0.9[4]. Since some metrics may be correlated and cause the problem of multicollinearity and overfitting (Harrell Jr, 2015; Jiarpakdee, Tantithamthavorn, & Hassan, 2018; S. Wang, Chen, & Hassan, 2018), we use Variance Inflation Factors (VIFs) to detect the collinearity among the metrics (Kuhn & Johnson, 2013). A high VIF value reflects an increase in the variance due to collinearity in the data. If a metric has a VIF value larger than 10, we remove the metric from the model (Kuhn & Johnson, 2013)[5].

Model Assessment Process

Our goal is not to predict post-release defect, but rather to study the explanatory power of the test smell metrics. Thus, we adopt three different model assessment techniques (probability of defect-proneness, Wald χ^2 test, and area under the curve; AUC) to understand the relationship between test smell and post-release defect.

First, we study the contribution of individual TEST_PRODUCT and TEST_PROCESS metric by looking at proportions of χ^2 for each metric relative to the total χ^2 of the model. χ^2 is a likelihood ratio test to identify how much a metric contributes to the model's fitness. The higher χ^2 indicates a higher explainability of the metric (i.e., more important) in the model (de Pádua & Shang, 2018; Harrell Jr, 2015). Finally, we use AUC, the area under the Receiver Operating Characteristics (ROC), to compare nested logistic regression to capture the relationship between the explanatory metrics and the source code's defect-proneness

[4]Redundancy analysis from the Hmisc R package.
[5]VIF analysis from RegClass R package.

file (Harrell Jr, 2015). AUC measures the fitness of the model. An increase in AUC when new metrics are added to the model indicates that the new model has a higher ability to capture the relationship and a better fitness (i.e., there is a correlation between the added metrics and defect-proneness, after controlling for the baseline metrics).

Second, we study the effect size of test smell metrics on the probability of defect-proneness (Moser et al., 2008; Shang, Nagappan, & Hassan, 2015). To quantify the effect, we set all of the model's metric values to their mean value and record the probability of defect-proneness. Then, we increase the value of the metrics in which we want to measure the effect (i.e., test smell metrics). For each subject metric, we increase the value by 125% and 150% of its mean value and re-calculate the probability of defect-proneness after the increase and report the percentage difference. A positive value indicates that increasing the metric's value increases the probability of the post-release defect. A negative value indicates that increasing the value of the metric decreases the likelihood of the post-release defect. The intuition behind the analysis is to understand which metric contributes more to the explainability of the software defects while controlling for other metrics (de Pádua & Shang, 2018).

Result. The explanatory power of test smell metrics in regression models. *Adding test smell metrics increase the AUC of the model by an average of 8.25% over the BASE model.* Table A.1 – A.6 in the appendix show the details of the regression models, where we show the additive effects of test smell features over the baseline metrics. We move the tables to the appendix to make the paper more concise. We show the proportion of χ^2 to understand the importance of including the metric on the model fitness. We show the AUC to understand whether our test smell metric contributes to a higher ability to capture the relationship on the post-release defect over baseline metrics. We find that in all of the models, the AUC increases by 5.1% over the baseline when adding TEST_PRODUCT metrics; and there is around 8.25% increase in AUC over the baseline when adding *both of the* TEST_PRODUCT and TEST_PROCESS metrics. Although the increase is small, we see a consistent result in all the studied systems except Wicket (as discussed in RQ1, Wicket experienced major refactoring in 2019, which may affect the modeling results). Our

results also show that adding TEST_PROCESS metrics have only a small increase in the model's explainability after considering the baseline metrics and TEST_PRODUCT. The potential reason may be that, as shown in RQ2, developers may remove or add a test smell as a by-product of other refactoring activities. Therefore, there may be noises in the TEST_PROCESS metrics. Lastly, for the proportion of χ^2, we find that the explainability of test smell metrics varies from system to system. However, we see that test smell metrics such as *Conditional Test Logic*, *Constructor Initialization*, *Exception Catch/Throw*, *Mystery Guest*, *Resource Optimism*, *Assertion Roulette*, *Eager Test*, and *Lazy Test* have higher explainability across the studied systems. In short, these test smells may have a higher correlation with the defect-proneness of source code files.

The effect size of test smell metrics on defect-proneness. *Most test smell metrics have minimal effect on defect-proneness.* The analysis mentioned above shows the explainability of the metrics but not the effect. Hence, we further study the effect of each test smell metric. Table A.7 – A.9 in the appendix show the effect size of the TEST_PRODUCT and TEST_PROCESS metrics on post-release defects for the studied systems. As discussed in the approach section, we measure the effect size by increasing individual test smell metrics while keeping all other metrics at the mean value. We find that the effect size and direction (i.e., positive or negative) of the effect vary from system to system. However, the effects of most test metrics on the defect-proneness are minimal (i.e., less than 1% increase in the probability of defect-proneness when 150% increases the value of the test smell metric). Compared to TEST_PROCESS metrics, TEST_PRODUCT metrics, in general, have a slightly larger positive relationship with source code defect-proneness. Among all test smell metrics, we find that *Exception Catch/Throw* and *Conditional Test Logic* show the highest positive relationship in the majority of the studied systems. The findings imply that more *Exception Catch/Throw* and *Conditional Test Logic* in a test case may lead to a higher probability of having a post-release defect in its corresponding source code file. The analysis result on *Exception Catch/Throw* also echoes our finding in RQ2. We found that developers are more likely to refactor *Exception Catch/Throw* when maintaining test code.

On the other hand, in RQ2, we only observed one commit that addressed *Conditional Test Logic*. Prior research (Peruma et al., 2019a) found that developers do not naturally think of *Conditional Test Logic* as a problem. Hence, future studies are needed to evaluate further the effect of this test smell on software quality. Finally, one possible reason for the high variability in effect size of the TEST_PROCESS metric compared to the TEST_PRODUCT metric could be that many of the test smells were removed as a by-product in the effort to improve test code maintainability (as found in RQ2). Thus, future research on test smell should consider those by-product removals and relocation when designing the study.

Table 4.6: The comparison of the area under (a ROC) curve for the studied systems. The model is trained using the system in the first column, and AUC is calculated using the system depicted in the remaining columns.

	Accumulo	Bookkeeper	Camel	Cassandra	Cxf	Flink	Groovy	Hadoop	Hive	Kafka	Karaf	Wicket	Zookeeper
Accumulo	0.73	0.61	0.52	0.68	0.62	0.63	0.59	0.60	0.53	0.66	0.59	0.59	0.60
Bookkeeper	0.62	0.87	0.62	0.64	0.58	0.64	0.71	0.54	0.58	0.61	0.61	0.58	0.51
Camel	0.60	0.65	0.75	0.63	0.68	0.64	0.51	0.61	0.52	0.66	0.49	0.69	0.62
Cassandra	0.67	0.66	0.57	0.78	0.69	0.64	0.55	0.65	0.55	0.59	0.53	0.67	0.61
Cxf	0.62	0.50	0.69	0.63	0.78	0.62	0.58	0.71	0.52	0.62	0.58	0.68	0.55
Flink	0.66	0.80	0.65	0.72	0.67	0.77	0.83	0.65	0.59	0.79	0.61	0.73	0.70
Groovy	0.60	0.63	0.53	0.66	0.63	0.64	0.97	0.58	0.51	0.65	0.60	0.57	0.63
Hadoop	0.60	0.63	0.53	0.66	0.63	0.64	0.97	0.58	0.51	0.65	0.60	0.57	0.63
Hive	0.61	0.62	0.67	0.62	0.62	0.64	0.81	0.61	0.67	0.61	0.64	0.67	0.56
Kafka	0.64	0.77	0.55	0.71	0.67	0.73	0.64	0.61	0.59	0.82	0.53	0.64	0.67
Karaf	0.54	0.59	0.54	0.59	0.53	0.57	0.49	0.63	0.54	0.58	0.82	0.60	0.65
Wicket	0.53	0.56	0.49	0.53	0.52	0.51	0.58	0.55	0.52	0.60	0.58	0.82	0.59
Zookeeper	0.51	0.58	0.57	0.57	0.58	0.56	0.73	0.71	0.53	0.52	0.61	0.55	0.83

The comparison of area under (a ROC) curve for the studied systems. *The cross-system AUC is lower than within-system AUC.* We further investigate whether different systems share a similar relationship between test smell and defect proneness (i.e., whether the models are applicable cross-systems). Table 4.6 shows the results of our cross-system AUC using combined (i.e., product and process) test smell features. In general, we find that the results of cross-system AUC are lower than the within-system AUC. In particular, some models trained on one system (e.g., Accumulo) perform worse when applied on some systems

(e.g., AUC is 0.53 when the model is applied on Camel) but are relatively better when applied on other systems (e.g., AUC is 0.67 when applied on Cassandra and Kafka). Our results show that while different systems have different development characteristics, some systems may have a more similar relationship between test smells and defect-proneness. Future studies are needed to further study effect of test smells across systems from different domains.

The studied test smell metrics increase the AUC of the model by an average of 8.25% over the BASE model. The test smell product metrics such as *Conditional Test Logic*, *Constructor Initialization*, *Exception Catch/Throw*, *Mystery Guest*, *Resource Optimism*, *Assertion Roulette*, *Eager Test*, and *Lazy Test* may have a higher correlation with the defect-proneness, while process metrics have little or no improvements to the model fitness. In short, most test smell metrics have minimal effect on defect-proneness, and different test smell has a different effect on the defect-proneness of source code files across the studied systems.

4.5 Threats to Validity

External Validity. The studied systems are all open source and implemented in Java, so the results may not be generalizable to all systems. To minimize the threat, we study systems that are large in scale, cover various domains, frequently used in commercial settings, and diversify the pool of test code under analysis based on the expertise of the developer. Even though our results are consistent among the studied systems, other developers/systems might exhibit a different awareness level about the test smells. Therefore, future studies must evaluate the results on additional systems and systems implemented in different programming languages.

Internal Validity. There may be confounding metrics that may affect the result of our logistic regression model. To mitigate this, we include baseline metrics, such as lines of test code, code churn, and two coupling metrics (i.e., source code to test code dependencies and test code to test code dependencies) in the model. Moreover, our model does not

indicate a causal relationship, but rather that there is a possibility of a relationship that may be further investigated in future research. Furthermore, our study aims to understand the relationship between test smell metrics to software post-release defects by studying the effect of test smells on post-release defects. Therefore, we build a logistic regression model to study the relationship between test smell and post-release defects, because logistic regression models provide better interpretability compared to more advanced machine learning model. Future studies investigating the effect of test smell on prediction performance should study just-in-time prediction and cross-system prediction.

Construct Validity. There may be false positives in the tool, *tsDetector*, that we used for identifying test smells. However, we found that false positive rate to be low in our manual classification. We found 12 false positives (4% false positive rate), which is consistent with the number reported in the prior study (Peruma et al., 2019a). Moreover, there may be biases in our manual classification on characterizing the commits that remove test smells. To minimize the biases, two authors independently inspect every commit and then merge the results. Furthermore, we examine all available software artifacts, such as commit messages, code changes, and bug reports. As for the reasons for removing test smells, many non-technical factors may play a role, such as a lack of knowledge and lack of time. However, in the bug report we analyzed, we did not find that developers mention such non-technical aspects that challenge test code's maintainability. Future studies should further dedicate studies on such non-technical factors. Moreover, while we find evidence in software artifacts that demonstrates developers' awareness of test smells, our manual classification serves only as a proxy measure for why developers removed test smells. Future research should involve presenting these removal instances to developers immediately after they occur to better understand the true reasons behind the removal of test smells. A recent paper Spadini et al. (2020) reported a severity threshold for *tsDetector* to make a recommendation when test smells are prevalent. Such results have a low impact on our results because we study test smells removal at a more general level, not only when there are too many tests smells. Finally, in our time series plot (RQ1), we plot the averaged test smell metrics of all systems. To ensure that one system does not overestimate the average, leading to false trends, we

verified that the individual systems' time series has the same trend as the average.

4.6 Implication & Contribution

Table 4.7 summarizes our findings for each research questions. We additionally discuss implications for each result.

Table 4.7: Summary of our findings and their implications.

Findings about how test smells evolve overtime	**Implications**
F.1 Although the total number of test smell increases over time, after normalizing by the total number of lines of test code, the test smell density remains relatively stable in most of the 12 studied systems.	**I.1** As software system evolves, test smell will likely co-evolve with amount of added test code. However, our results suggest that developers may allocate some resources in maintaining test code that results in removal of test smells.
Findings about test smell <u>awareness</u> refactorings	**Implications**
F.1 *Sleepy Test* & *Exception Catch/Throw* are the two test smells that developers directly address..	**I.1** Our results may help future research and tool builders to focus on these two test smells for a better recommendation support on addressing test smells.
F.2 Developers sometimes refactor test smell to remove verbose statements. In particular, developers may use better assertion style to remove test smells.	**I.2** There are numerous testing frameworks that offer distinct assertion syntax. However, due to lack of experience and knowledge, developers may sometimes resort to verbose assertions. Future research should investigate refactoring recommendation using better assertion statements.
Findings about test smell <u>unawareness</u> refactorings	**Implications**
F.1 58 out of 292 (20%) commits relocated test smells to another test case after some refactoring and maintenance activities. In such cases, developers pay attention to test code reusability and duplication instead of addressing test smells. Subsequently, we find that in some cases relocation can diffuse the impact of test smells (e.g., relocated to a utility file).	**I.1** Although the maintenance test code has become a prominent task in recent years, for the most part, test smell is not the reason for refactoring, and developers may not pay attention to addressing test smells.
F.2 70 out of 292 (24%) commits remove test smells while working on other maintenance tasks.	**I.2** Test smells are inherent problems, which may hinder test design and comprehension. However, our result shows that developers may not be aware of the test smells. Many test smells are indirectly removed when developers deal with bug fixing or feature enhancement.
Findings about other maintenance activities that removed test smells	**Implications**
F.1 Most test smells are removed due to test code deletion (33%). We find that as test code evolves, there may be substantial instances of ad-hoc manual test code deletions caused by redundant or obsolete test code, which remove test smells as a side-effect.	**I.1** Our result suggests that developers often manually maintain test code by deleting duplicate or obsolete test code, which may be time consuming. Future studies should support the detection of refactoring opportunities or even conduct automated refactoring to reduce maintainability efforts.
F.2 Developer tends to disable (i.e., commenting out or ignoring) test case (2%) to make a test pass.	**I.2** Future studies should further investigate the causes for disabling test cases, and whether disabled test cases become technical debts in the systems (e.g., forget to re-enable) (Cunningham, 1993; Pham & Yang, 2020a).
Findings about relationship between test smell and software quality	**Implications**
F.1 Test smell metrics complement traditional metrics in explaining post-release defects, even though the addition is small (an average of 5.8% increase in AUC). **F.2** The test smells such as *Conditional Test Logic*, *Constructor Initialization*, *Exception Catch/Throw*, *Mystery Guest*, *Resource Optimism*, *Assertion Roulette*, *Eager Test*, and *Lazy Test* have a higher defect explanatory power in the model. **F.3** The effect sizes of most test smell metrics on defect-proneness are small. Among all test smells, *Conditional Test Logic* and *Exception Catch/Throw* have the largest effect on a post-release defect.	**I.1** Our result suggests that test smell metrics have a certain correlation with post-release defect, even though the correlation (i.e., improvement in model fitness) is not large. **I.2** Future studies on test smells may focus more on the above-mentioned test smell due to their higher explainability in post-release defect in the model. **I.3** Future studies should further investigate the effect of *Conditional Test Logic* and *Exception Catch/Throw* on software quality and help practitioners prioritize their effort on addressing test smells.

4.7 Chapter Summary

First and foremost, the primary value of our research work comes from recognizing the importance of understanding why developers remove test smells and the mechanisms in which they are addressed. We believe this is a necessary corequisite to validate current perception of test smells towards developing a more useful refactoring recommendation tool. Without such knowledge, future studies may progress to propose new test smells and detection tools with minor applicability in the wild and may even hamper software maintenance effort. To that end, we attempt to tackle this problem in three folds. First, we find that developers may allocate resources in the maintenance of test code. The test smell density decreases over time, even though the total number of test smell increases in the software systems. Second, we find that developers are more likely to address a subset of test smells (i.e., *Exception Catch/Throw* and *Sleepy Test*) and the rest were usually removed indirectly as a side-effect of accomplishing other non-trivial maintenance tasks related to fixing bugs or change in feature requirements. Similarly, we identify other code changes besides refactoring that relocate, diffuse, delete, disable and revert test code that caused the removal of test smells. Finally, we apply regression models to understand the relationship between test smell metrics and post-release defect. After controlling for baseline metrics (i.e., LOC, code churn, pre-release defect, and coupling in test code), we find that test smell metrics provide additional defect explanatory power, although the increase is small. Our model also finds that test smells such as *Exception Catch/Throw* and *Conditional Test Logic* have a larger effect on post-release defect. In summary, our study highlights that developers may allocate resources on maintaining test code, but they often do not address test smells. However, we find that some test smells do have some relationship between post-release defect. Future studies are needed to better assist developers with prioritizing the resources to address test smells and refactoring test code.

Chapter 5

Demystifying test annotation maintenance in the wild

Since the introduction of annotations in Java 5, the majority of testing frameworks, such as JUnit, TestNG, and Mockito, have adopted annotations in their core design. This adoption affected the testing practices in every step of the test life-cycle, from fixture setup and test execution to fixture teardown. Despite the importance of test annotations, most research on test maintenance has mainly focused on test code quality and test assertions. As a result, there is little empirical evidence on the evolution and maintenance of test annotations. To fill this gap, we perform the first fine-grained empirical study on annotation changes. We developed a tool to mine 82,810 commits and detect 23,936 instances of test annotation changes from 12 open-source Java projects. Our main findings are: (1) Test annotation changes are more frequent than rename and type change refactorings. (2) We recover various migration efforts within the same testing framework or between different frameworks by analyzing common annotation replacement patterns. (3) We create a taxonomy by manually inspecting and classifying a sample of 368 test annotation changes and documenting the motivations driving these changes. Finally, we present a list of actionable implications for developers, researchers, and framework designers.

Earlier version of this chapter was published in the 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021). 12 pages. (D. J. Kim, Tsantalis, Chen, & Yang, 2021)

5.1 Introduction

Modern software systems are becoming more complex due to the ever-growing demands from customers. To ensure that the software quality remains on par with consumer expectations, testing has become a pivotal role in software development. Developers rely on testing to verify the quality of every code change and provide an indication on whether the software can be released to production (Ali et al., 2019).

To increase the effectiveness of testing, developers need to maintain and improve test code continuously. Similar to source code, test code may also contain design issues that hinder the quality. For example, prior studies have found that the results of some test cases can be unreliable (i.e., flaky tests) due to bugs in test code (Lam, Godefroid, Nath, Santhiar, & Thummalapenta, 2019b; Luo, Hariri, Eloussi, & Marinov, 2014b). To that end, developers have begun to notice a recurring design problem in test code and coined the term test smells (Van Deursen et al., 2001) as an indicator of design problems in tests. Since its inception, researchers have shown that test smells are prevalent in software systems (Peruma et al., 2019a), negatively affect software maintainability and comprehension (Bavota et al., 2012; Bavota et al., 2015), and may impact software quality in terms of post-release defects (Spadini et al., 2018).

The introduction of annotations in Java 5 has driven annotation as a critical component of the many Java-based frameworks, influencing how developers to design and implement software Z. Yu, Bai, Seinturier, and Monperrus (2019). Even in software testing, frameworks such as JUnit, TestNG, and Mockito have all adopted annotations as critical ingredients in test design and implementation. A prior study (Zerouali & Mens, 2017) has found that JUnit4 is one of the most widely utilized testing frameworks for Java-based systems, and test annotations (e.g., @Test) are also one of the most widely used annotations in Java

Table 5.1: A brief overview on commonly used JUnit4 annotations.

Annotations	Annotation Location	Description of Commonly used JUnit Annotation
@Rule	Field	@Rule provides a mechanism to enhance tests by running some code around a test case execution, which is similar to fixture and teardown.
@Parameterize	Field/Method	Test case annotated with @Parameterize can be invoked by using a predefined input (i.e., parameterized test inputs) and expected output.
@Test	Method	@Test indicates that the annotated test code should be executed as a test case. @Test takes optional parameters, such as Timeout to indicate that the test should finish within a given time, or exception to indicate that the test should throw an exception.
@Before/@After	Method	@Before indicates that the annotated test code should be executed as a precondition before each test case (i.e., Database setup). Similarly, @After indicates the execution of the annotated test code as a postcondition after each test case.
@BeforeClass/@AfterClass	Method	@BeforeClass and @AfterClass are similar to @Before and @After annotation types, but indicate the annotated test code to only execute once (i.e., before or after the test class is invoked).
@Ignore	Method/Class	@Ignore indicates that the annotated test case should not execute.
@Category	Method/Class	@Category provides a mechanism to label and group tests, giving developers the option to include or exclude groups from execution.
@Test(timeout=X)	Method/Class	A test will fail, if its execution takes longer than the value X specified in timeout.

development. Table 5.1 provides an overview of the commonly used test annotations in JUnit4. Although the test annotations may be different across testing frameworks (e.g., TestNG or JUnit5), in general, they provide similar functionalities.

Despite the importance of test annotations, most prior research on test maintenance has only focused on general test design and test assertions (Athanasiou et al., 2014; Bavota et al., 2012; Bavota et al., 2015; Garousi & Küçük, 2018; Greiler, Van Deursen, & Storey, 2013; Greiler, Zaidman, Van Deursen, & Storey, 2013; Junior et al., 2020a; Qusef et al., 2019) and has not considered the peculiarity of test annotations. Therefore, in this paper, we present the first empirical study on how developers leverage test annotations in the wild to maintain the high quality of test code (e.g., readability, test flakiness, test performance, obsolete test). We first extended the state-of-the-art refactoring mining tool, RefactoringMiner 2.0 (Tsantalis, Ketkar, & Dig, 2020b), to detect annotation additions, removals, and modifications. We study the collected annotation changes both quantitatively and through manual classification by answering three research questions:

RQ1: How common are test annotation changes? Test annotation changes are 26.5% more common than regular test refactorings such as renames and type changes. Despite their popularity, there is negligible tool support (e.g., antipattern detection, annotation change suggestions, and annotation API usages) for test annotation changes.

RQ2: How are test annotations changed in the wild? We quantitatively study frequent test annotation changes. We find that developers update test annotations more frequently than annotation additions and removals. Moreover, test annotation migration across testing frameworks and within a different version of the same framework is also common.

RQ3: Why do developers change test annotations? We conduct a manual classification to uncover test annotation usage and misusage, and how developers bypass the limitations of test annotations. Our findings highlight potential future directions on helping developers improve test maintenance and detect potential issues in test code.

In summary, our findings provide actionable implications for three groups of audiences:

(1) **Researchers**: We open an avenue for further research directions on detecting misuses and test smells related to test annotations and their relation with other aspects of software development (i.e., quality, maintainability, and performance improvements). We also highlight potential directions on automated test code refactoring by leveraging test annotations.
(2) **Developers**: Our findings reveal the usage of test annotations in an ad-hoc manner by some developers. A number of test cases is temporarily disabled until a fix is found; however, the disabled tests are not re-enabled after the fix. Moreover, in several cases, developers are unaware of the features offered by testing frameworks, and thus apply suboptimal custom solutions. These findings indicate the need to educate developers with the best testing practices and provide recommendation tools to help developers apply the appropriate test annotations where needed.
(3) **Framework Designers**: We find cases where developers try to bypass the current limitations of test annotations (i.e., fixture configuration) and provide suggestions for framework designers on improving the flexibility of test annotations.

5.2 Method for Detecting Changes in Test Annotations

RefactoringMiner Extension. In order to detect annotation additions, removals and modifications, we extended the state-of-the-art refactoring mining tool, RefactoringMiner 2.0 (Tsantalis et al., 2020b). We selected this tool for the following reasons:

(1) It operates at commit level, allowing us to obtain annotation changes at the finest granularity level of software evolution (i.e., commits).

(2) It can detect refactoring operations, allowing us to include annotation changes for refactored program elements (i.e., methods with changes in their signatures, moved/renamed classes and fields) in addition to non-refactored program elements. This makes our dataset more complete and our findings more reliable.

(3) It has the highest precision (96.6%) and recall (94%) among other refactoring mining

and AST diff tools, allowing us to have an accurate dataset of annotation changes with a very small number of false positives and false negatives.

(4) It has the fastest execution time among other refactoring mining tools, allowing us to scale up our data collection for the entire commit history of large projects with over 20K commits.

Using the RefactoringMiner API, we obtain the pairs of program elements (i.e., type, method and field declarations), which have been matched between the currently analyzed commit and its parent in the directed acyclic graph that models the commit history of git-based version control repositories. The pairs of matched program elements may have identical signatures (e.g., a pair of methods with identical names, parameter and return types), or may have different signatures due to refactoring operations (e.g., Rename Method, Change Parameter Type, Add/Delete Parameter).

Java annotations are used in three different forms:

- **Marker** annotations without member value pairs: `@TypeName`.
- **Normal** annotations with a list of member value pairs: `@TypeName(name1=value1,name2=value2,..)`, where names are `SimpleName` AST nodes and values are `Expression` AST nodes.
- **Single Member** annotations with a single member value: `@TypeName(Expression)`, where the member name is omitted (i.e., `@foo(bar)` is equivalent to the normal annotation `@foo(name=bar)`).

We consider two annotations as equal if they have the same `TypeName`, and the same member value pairs regardless of their order. Let us assume that for a given pair of matched program elements A_p is the annotation set of the program element in the parent commit, and A_c is the annotation set of the matched program element in the child commit. Then, the added annotations are computed as $A^+ = A_c \setminus (A_p \cap A_c)$. The removed annotations are computed as $A^- = A_p \setminus (A_p \cap A_c)$. The pairs of modified annotations are computed as

$A^{\sim} = \{(a_p, a_c) | a_p \in A_p \wedge a_c \in A_c \wedge a_p.\texttt{TypeName} = a_c.\texttt{TypeName} \wedge a_p.\texttt{MemberValuePairs} \neq a_c.\texttt{MemberValuePairs}\}$.

To evaluate the precision and recall of our RefactoringMiner extension, we extended the oracle used in (Tsantalis et al., 2020b), which contains true refactoring instances found in 536 commits from 185 open-source GitHub projects, with instances of Annotation Additions/Removals/Modifications for four different program elements, namely type, method, field, and parameter declarations. To compute precision, an author of the paper manually validated 638 annotation change instances reported by our RefactoringMiner extension. To compute recall, we need to find all true instances of annotation changes. We followed the same approach as in (Tsantalis et al., 2020b) by executing a second tool, namely GumTree (Falleri, Morandat, Blanc, Martinez, & Monperrus, 2014), and considering as the ground truth the union of the true positives reported by RefactoringMiner and GumTree. GumTree takes as input two abstract syntax trees (e.g., Java compilation units) and produces the shortest possible edit script to convert one tree to another. We used all *Insert* and *Delete* edit operations on Annotation AST nodes to extract annotation changes and report them in the same format used by RefactoringMiner. Table 5.2 shows the number of true positives (TP), false positives (FP), and false negatives (FN) detected/missed by our RefactoringMiner extension. The overall precision is 99.7% and the recall is 98.7%.

Table 5.2: Precision and recall of our extended version of RefactoringMiner.

Change Type	TP	FP	FN	Precision	Recall
Add Method Annotation	312	1	7	99.7%	97.8%
Remove Method Annotation	97	1	0	99%	100%
Modify Method Annotation	19	0	0	100%	100%
Add Parameter Annotation	29	0	0	100%	100%
Remove Parameter Annotation	3	0	0	100%	100%
Modify Parameter Annotation	2	0	0	100%	100%
Add Field Annotation	47	0	1	100%	97.9%
Remove Field Annotation	17	0	0	100%	100%
Modify Field Annotation	7	0	0	100%	100%
Add Class Annotation	52	0	0	100%	100%
Remove Class Annotation	20	0	0	100%	100%
Modify Class Annotation	31	0	0	100%	100%
Overall	636	2	8	99.7%	98.7%

Table 5.3: An overview of the studied systems (from 2015 to 2020).

Systems	Total Test LOC (2015 → 2020)	No. Test Method (2015 → 2020)	No. Test Class (2015 → 2020)
Ambari	125K → 273.8K	2,471 → 5,753	501 → 999
Camel	562K → 787K	12,884 → 18,693	6,713 → 8,961
Cassandra	44.3K → 189K	969 → 4,515	217 → 626
Druid	45K → 307K	773 → 5,818	199 → 1,148
Flink	79K → 437K	1,416 → 9,199	412 → 2,150
Hadoop	480K → 914K	9,269 → 17,610	1,798 → 2,954
Hbase	185K → 359K	2,843 → 5,861	660 → 1,476
Hive	124K → 323K	2,572 → 7,541	473 → 1,182
Ignite	261K → 99K	4,146 → 2,286	1,285 → 529
Kafka	2.9K → 191K	77 → 6,059	24 → 688
Openfire	2.2K → 8.3K	84 → 361	25 → 51
Storm	3.7K → 47K	118 → 1,134	35 → 277
Total	1916K → 3939K	37,622 → 84,830	12,342 → 21,041

5.3 Studied Systems

We choose the studied systems by following three selection criteria. First, we selected the top 1,000 Java projects on GitHub ordered by popularity (i.e., stargazer count). We also made sure that the repositories are not forks. Second, we discarded projects that are below 90 percentile in terms of size (i.e., lines of code), repository popularity (i.e., stars) and the number of commits. Finally, we discarded inactive repositories that did not have any commits in 2020. We ended up with 12 systems, i.e., Druid, Hadoop, Cassandra, Storm, Flink, Hbase, Camel, Hive, Openfire, Ambari, OrientDB and Kafka. These studied systems cover different domains, ranging from distributed databases, stream processing frameworks, message brokers, and groupchat servers. Table 5.3 shows an overview of the studied systems.

Our study focuses on test annotation usage, but there may be some non-test-related annotations in test classes. Hence, we set off to understand what are the common testing frameworks or libraries from which test-related annotations are used. We mined all annotation usages in the versions released in 2015 and 2020, respectively, for the 12 studied systems. In particular, we analyzed all test files that have a *".java* extension and "[Tt]est(s*)" as prefix or suffix in their name. We manually verify the build configuration files (e.g., Maven or Gradle build file) of the studied systems to use the default heuristic specified by Maven/Gradle plugin to identify test files. After collecting the annotation usages, we manually study them and identify their corresponding framework. Table 5.4 summarizes

Table 5.4: Use of Annotations from Different Frameworks in Test Code Released in 2015 and 2020, respectively.

Framework Type	Frequency	Proportion (%)
Testing Framework	**32,900 → 102,395**	**72.6% → 65%**
JUnit	31,256 → 101,047	69% → 64%
TestNG	1,644 → 1,348	3.6% → < 1%
Mocking Framework	**260 → 1,640**	**< 1% → 1%**
Mockito	236 → 935	< 1% → < 1%
PowerMock	24 → 256	< 1% → < 1%
EasyMock	0 → 449	0% → < 1%
Java lang annotations	**8,579 → 16,125**	**19% → 10%**
Custom Annotation	**2,738 → 36,300**	**1.7% → 23%**
Spring Framework	**442 → 551**	**1% → < 1%**
Other Libraries E.g., Google, JavaX	**410 → 17,369**	**1% → 1%**

the annotation usage of different frameworks in the 12 studied systems. We find that JUnit annotations are the most commonly used annotations in test code, accounting for 69% of the mined annotations in 2015 and 64% in 2020. TestNG is a less commonly used testing framework, accounting for 3.6% of the mined annotations in 2015 and 1% in 2020. Our finding shows that developers in the studied systems are migrating away from TestNG. We also found annotations from frameworks used for test mocking (i.e., Mockito, PowerMock and EasyMock), the Spring framework, and other non-test-related libraries. The annotation usage of testing frameworks, such as JUnit and mocking frameworks, increases significantly over the years. However, their percentages decrease due to the increasing use of custom annotations. In section VI, we discuss some of the custom annotations related to testing.

To collect annotation changes, we run our RefactoringMiner extension on every commit that modified at least one Java test file between 2015-2020 for the 12 studied systems. We only keep changes on test-related annotations (i.e., from JUnit, TestNG and mocking frameworks) and discard the rest. In total, we mined 109,460 test-related annotation changes in the commit history of the 12 studied systems from 2015 to 2020.[1]

5.4 Results

In this section, we conduct a quantitative study to understand the prevalence of test annotation usages and change patterns. In the last RQ, we conduct a manual classification to understand the reasons that developers change annotations and how test annotation helps improve test maintainability.

5.4.1 RQ1: How common are test annotation changes?

As a stepping stone to understanding how developers leverage annotations, we examine how frequently test annotations are changed compared to common source code changes (i.e., renames and type changes) at the same program element level.

Approach. We study how frequently developers change test annotations. To provide some comparative statistics, we show the prevalence of test annotation changes compared to common source code transformations (i.e., renames and type changes) at the same program element level (i.e., class, method and field declaration). In particular, we compare test annotation changes at the method level with Rename Method and Change Return Type, at field level with Rename Field and Change Field Type, and at the class level with Rename Class. Such a comparison is attainable because all compared changes are performed on the same kind of program elements. We used the tool implemented by Ketkar, Tsantalis, and Dig (2020b) to detect renames and type changes. Ketkar et al. (2020b) report an average precision of 99.7% and a recall of 94.8% for type change detection, and an average precision of 99% and recall of 91% for rename detection, which is very close to the precision/recall values reported in Table 5.2, allowing for a fair comparison annotation change and refactoring practices.

Result. Table 5.5 compares the prevalence of test annotation changes with that of the

Table 5.5: Mined Test Code Changes in 82,810 Commits.

	Field		Method		Class		Total	Num.
		Commit		Commit		Commit		
Annotation Changes	396	0.075	18,472	3.52	5,068	0.97	23,936	5,249
Refactoring	7,125	0.40	9,657	0.53	2,136	0.12	18,918	17,914
Δ % Percentage	-94.4%	-81.3%	+91.3%	+564.2%	+137.2%	+708.3%	+26.5%	-70.7%

refactoring changes in test code from the 82,810 commits. As shown in Table 5.5, the number of test annotation changes is comparable to the number of test refactorings (i.e., 26.5% difference). Test annotation changes are performed at a method and class level more than renames and type changes, i.e., 91.3% and 137.2% more, respectively. However, at the field level, test annotation changes occurred less than renames and type changes. Despite the popularity of test annotation changes, little tool support exists for test annotations compared to common code transformations such as renames and type changes.

We find that much fewer commits modify test annotations than those that perform renames and type changes. Out of the 82,810 commits, 5,249 commits (6%) modify test annotations, and 17,914 (21%) perform code transformations such as renaming and type changes. Once normalized by the number of commits, test annotation changes at class and method level are performed much more frequently than renames and type changes. This shows that test annotation changes at class and method levels are more concentrated in fewer commits, suggesting that annotations may be associated with dedicated maintenance activities. In RQ3, we will further discuss ways annotations are utilized in the maintenance of test code.

> Test annotation changes are comparable to renames and type changes at the same program element level and are even more frequently applied at the method and class level. Despite the popularity, there is currently negligible tool support (i.e., antipattern detection, annotation change suggestions, and annotation API usages) for test annotation changes.

5.4.2 RQ2: How are test annotations changed in the wild?

We examine what are the common test annotation change patterns in the wild. Studying predominant annotation changes reveals frequent maintainability activities developers perform through test annotation changes as software evolves. Such insights act as stepping stones for our subsequent manual classification on the motivations and challenges behind test annotation changes.

Table 5.6: Quantitative Analysis: Top three highest frequency of annotation addition, removal and modification.

Addition	Freq.	Removal	Freq.	Modification	Freq.
Field Level					
@Mock	147	@Rule	57	@Mock	23
@Rule	42	@Mock	31	@Parameter	12
@Parameter	23	@ClassRule	19	@Parameterized	4
Method Level					
@Ignore	1362	@Ignore	968	@Test	6874
@Before	1238	@Before	482	@Parameterized	94
@After	584	@BeforeClass	293	@Parameters	25
Class Level					
@RunWith	770	@Ignore	318	@Category	1506
@Category	734	@RunWith	306	@RunWith	326
@Ignore	482	@Category	201	@PrepareForTest	91

Table 5.7: Quantitative Analysis of annotation replacements.

Granularity	Annotation Changes	Freq.	Total
Field Level			**27 (2.4%)**
JUnit	@Rule ↔ @ClassRule	17	
JUnit4 → JUnit5	@ClassRule/@Rule → @RegisterExtension	4	
	@ClassRule → @Container	3	
	@Rule → @TempDir	3	
Method Level			**1,007 (91%)**
JUnit	@BeforeClass/@AfterClass → @Before/@After	332	500 (45%)
	@Before/@After → @BeforeClass/@AfterClass	148	
	@Before/@After → @After/@Before	11	
	@Parameters ↔ @Parameterized	6	
	Timeout=X in @Test → @Timeout	2	
	@BeforeClass → @AfterClass	1	
JUnit4 → JUnit5	@Before → @BeforeEach	216	313 (28%)
	@After → @AfterEach	43	
	@BeforeClass → @BeforeAll	23	
	@AfterClass → @AfterAll	19	
	@Ignore → @Disabled	12	
TestNG → JUnit	@BeforeMethod/@AfterMethod → @Before/@After	108	142 (13%)
	@Test(Enabled=False) → @Ignore	30	
	@BeforeTest → @Before	4	
Custom ↔ JUnit	@TestTag → @Category	25	27 (2.5%)
	@Category(PerformanceTest.class) → @PerformanceTest	2	
TestNG	@BeforeTest → @BeforeClass/@BeforeMethod	6	21 (1.9%)
	@AfterTest → @AfterClass/@AfterMethod	9	
	@AfterClass → @AfterMethod	3	
	@AfterMethod → @BeforeMethod	1	
	@BeforeClass → @BeforeMethod	2	
JUnit4 → SpringBoot	@Category → @IntegrationTest	4	4 (0.4%)
Class Level			**70 (6.3%)**
JUnit4 → JUnit5	@Ignore → @Disabled	46	68 (6.3%)
	@RunWith → @ExtendWith	17	
	@Ignore → @Category	5	
JUnit4 → SpringBoot	@Category → @IntegrationTest	2	2 (<1%)
Total		**1104**	

Result. We present the quantitative analysis on test annotation change patterns from two aspects. First, we present the raw change patterns based on three types of changes: addition, removal and modification. The three types of changes are the direct output from our RefactoringMiner extension. Table 5.6 lists the top three annotation changes per change type at three program levels (i.e., field, class and method). We observe that modification has a strong prevalence at a method and class level across the three types of changes. Developers frequently update parameters of the @Test (e.g., timeout=X) and @Category annotations. In general, we notice that developers frequently change the test annotations from mocking frameworks, i.e., add @Mock and @RunWith (20% of PowerMockRunner, 14% of MockitoJUnitRunner). Considering the low prevalence of mocking frameworks in tests (around 1% as shown in Table 5.3), this suggests that the mocking frameworks are frequently updated as code evolves. We also find that developers frequently add and modify @Parameter, @Parameterized and @RunWith (50% of Parameterized) annotations, suggesting that expanding test input and diversifying test execution settings is commonly leveraged to facilitate code evolution.

We also observe a high prevalence of adding and removing the @Ignore annotation at both method and class level (i.e., disabling and enabling test cases and test classes). This suggests that technical debt may occur in test code evolution. Lastly, we observe a large number of modifications for the @Category annotation at the class-level to organize test classes into groups, and a diverse set of fixture additions and deletions (i.e., @Before, @After, @BeforeClass).

Furthermore, we perform an in-depth analysis to reveal a composite change pattern, i.e., an annotation replacement. A replacement @X→@Y occurs when annotation @X is removed and @Y is added on the same program element and commit. Annotation replacements may happen, within one testing framework, or between different testing frameworks. Replacements show that as software evolves, the original test annotation (or framework) does not satisfy the testing needs. Therefore developers may look for alternatives. However, such alternatives are not directly provided or are hard-to-achieve in the current framework. Hence, developers need to compromise with workarounds or adopt another framework.

Mining annotation replacements is straightforward, based on the output of our RefactoringMiner extension. Specifically, for each commit, we match pairs of removed and added annotations on the same program elements (i.e., fields, methods, and classes) and ensure that these pairs involve different annotation types.

In total, we mined 1,104 replacements from all mined annotation changes. Table 5.7 shows the frequencies of different replacement patterns. Most (91%) of the replacements are at the method level, and 45% of the replacements are switching between JUnit fixtures. For example, developers replace @BeforeClass/@AfterClass with @Before/@After or vice versa to configure the setup and tear down phases at the test class or test case level. Similarly, developers replace @Rule with @ClassRule at field level to expand the impact of a rule to the entire class. At all program element levels, we notice that many replacements occur due to migrations, i.e., between different testing frameworks, or from JUnit4 to JUnit5. Interestingly, we observe a few replacements between different frameworks, which are not due to migrations. Developers may find a similar test annotation in SpringBoot more suitable for a particular development need than the JUnit @Category annotation. Another common case is to replace custom annotations with the JUnit ones. Developers may define custom annotations for particular needs as JUnit may not yet support the desired features, or developers may not be aware of such support by JUnit. When the developers become aware of the unused JUnit features, or the desired features are shipped in the next JUnit releases, they tend to replace their custom annotations.

Our study shows that developers modify test annotations more frequently compared to additions and removals. Annotations from mocking frameworks are commonly changed despite their low prevalence. Further analysis of annotation replacements shows that they commonly occur for migrating to newer framework versions or other frameworks.

5.4.3 RQ3: Why do developers change test annotations?

Our goal is to provide suggestions to researchers, practitioners, and framework designers on opportunities to improve test annotations. We derive a taxonomy of test annotation

changes representing distinctive test maintenance efforts. We believe our taxonomy will provide insights on the maintenance of test annotations and how to improve test quality.

Approach. We manually study and understand the reasons that developers change test annotations by analyzing all available software artefacts: including issue ID from the Jira bug report, GitHub's pull request/issue tracking, commit messages and the code changes. Some commit messages contained sufficient information to understand the reason behind test annotation changes. Such commit messages may include keywords like *"Increase timeout"* (e.g., *Hadoop - e4c3b52*).

In particular, our manual classification is composed of the following phases:

Phase I: We use stratified random sampling, with a 95% confidence level and 5% confidence interval to acquire 368 annotation change samples from the test annotation changes identified by our RefactoringMiner extension. We use a 95% confidence level because it provides a high level of certainty that the true population parameter falls within the specified range. We adopted stratified random sampling to sample each studied system independently to reduce sampling error when a sub-population within the overall population varies (Zhao et al., 2019).

Phase II: To create the taxonomy for the test annotations, we first classified the changes at a high level based on the annotation type (e.g., @Ignore). Then, the first two authors of the paper (A1 an d A2) independently derived an initial list of the reasons behind annotation changes by manually inspecting the relevant commit messages, test source codes, and bug reports.

Phase III: Authors A1 and A2 unified the derived reasons and compared the assigned reason for each annotation change. Any disagreements were discussed until a consensus was reached. We used Cohen's Kappa inter-rater agreement to measure the degree of agreement between the two authors (Cohen, 1960). Cohen's Kappa considers a scenario where the agreement between two authors is purely by chance. The inter-rater agreement of the coding process has a Cohen's kappa of 0.91, indicating almost a perfect agreement level (Viera, Garrett, et al., 2005).

Result. Table 5.8 shows the derived taxonomy of the reasons that developers changed the

JUnit annotations (upper half of the table) and the annotations from other frameworks (lower half of the table) that we found in the sample. To encourage the replication of our results, we have made the dataset available[1]

JUnit Test Annotation Changes

@Ignore (32%). @Ignore is the most frequently changed test annotation (mostly added). Developers often use this annotation to temporarily disable the execution of tests when there are software bugs or flaky tests. Developers may also bypass test failures caused by recent code changes during a feature addition that breaks a test. For example, in *Hive* (7f4a3e17), the developer ignored the failing test code due to breaking changes during feature addition to pass the test temporarily. Other instances of adding @Ignore are due to dependencies with external libraries. For example, developers disabled a test while waiting for a new software version (e.g., JDK update). However, developers may also add @Ignore to replace automated testing with manual testing when the test code requires manual startup. Although developers frequently ignore tests to facilitate maintenance difficulties, this practice may become ad-hoc and affect code quality. We found instances where developers use @Ignore to pass failing tests without fixing the issue in the code. For example, in *Druid* (da32e1ae), the developer disabled a test due to unknown failure. Later on, the bug persisted, but the test was enabled, and the issue was closed. Similarly, in *Camel* (8ba68e34), the developer disabled a test due to external dependencies during feature addition. However, the ignored test is never enabled. We further conducted an exploratory investigation to see whether the ignored flaky tests in Table 5.8 are fixed and later enabled. We find that developers often do not find the root cause of test flakiness and ignore the test in the entirety of software evolution, indicating that ignored tests persist and are often forgotten.

> As @Ignore becomes a common way to bypass challenges in test maintenance, it may become ad-hoc and a source of technical debt.

1. https://github.com/SPEAR-SE/TestAnnotationMaintenance_Data

Table 5.8: manual classification: Taxonomy of Annotation Changes.

Annotation Type	Motivation	Frequency
	JUnit Test Annotations Changes	
Ignore		**115**
Bugs	Adding @Ignore to bypass test failure caused by bugs in test/source code.	42
Flaky test	Adding @Ignore to disable flaky tests.	33
External dependency	Adding @Ignore when an external dependency needs to be manually configured (e.g., database), or the developers wait for a new release of an external dependency to resolve an issue (e.g., JVM).	20
Feature addition/improvement	Adding @Ignore to disable tests that are related to incomplete features/code changes.	19
Timeout		**51**
Relax timeout	Increasing @Test(timeout) thresholds to accommodate slow cluster, slow machine or slow tests.	23
Deadlock detection	Adding @Test(timeout) to help detect deadlocks (i.e., if tests do not finish within the specified time, there may be a deadlock).	15
External resource retrieval	Adding @Test(timeout) to complement tests that retrieve an external resource. For example, without a timeout, the test may fail due to `NullPointerException` and suppress the actual fault (e.g., resource unavailability).	6
Perf. regression detection	Adding @Test(timeout) to ensure that the test finishes on time for detecting performance regression.	6
Ad-hoc timeout removal	Removing @Test(timeout) completely as tests become too slow instead of relaxing the timeout.	1
Fixture		**35**
Reset fixtures	Replacing @BeforeClass with @Before to reset fixtures for each test case (e.g., for bug fixing or test case isolation).	14
Improve test speed	Replacing @Before with @BeforeClass to improve test time by removing unnecessarily repeated fixture initialization.	10
Inflexible configuration	Removing or changing fixtures since they are not configurable per test case (e.g., @Before method runs for every test case, while @BeforeClass runs only once before a test case).	6
Maintainability	Adding fixtures to remove duplicate initialization in the test code for better maintainability.	1
Category		**37**
Test prioritization	Adding @Category to group tests based on their speed/size to detect failures more quickly (e.g., run faster tests first).	33
Ignore tests	Adding @Category in addition to @Ignore to organize ignored tests for future maintainability.	4
Parameterized		**31**
Increase test coverage	Adding or changing @Parameterized to increase coverage for failing corner cases, or newly added features.	16
Refactor test code	Adding @Parameterize to refactor tests to improve maintainability (e.g., share common test inputs or test code).	7
Parallelize tests	Adding @Parameterized and use a thread pool to run tests in parallel and speed up test execution.	4
Add debugging messages	Changing @Parameterized parameter to include optional messages for improved debugging.	3
Slow test	Removing @Parameterized to improve test execution time. In JUnit4, @Parameterized is limited to class level. If only subsets of tests use parameterized annotations, then it may increase test execution time.	1
Expected Exception		**18**
Adjusting exception handling	Changing between different exception handling mechanisms (i.e., *JUnit3* Try with fail, *JUnit4* @Rule, *JUnit5* Assertions.assertThrows, *JUnit4* @ExpectedException, or even custom expected exception)	14
Test Driven Development	Adding expected exception to complement test-driven development by making the test pass with known exceptions until the feature implementation is done.	2
Exception too general	Changing expected exception from a general exception type to a more specialized one. For example, @ExpectedException(GenericException) can pass the expected exception test, but does not provide details about the actual exception type.	2
Rule		**14**
Refactor via @Rule	Adding built-in or custom (e.g., extract duplicate fixture) @Rule to improve test code maintainability.	14
Fixed Test Order	Adding @FixMethodOrder from *JUnit4* to enforce deterministic test orders and fix flaky tests.	**5**
	Other Types of Test Annotations Changes	
Migration		**21**
JUnit4 to *JUnit5* migration	Manual migration from *JUnit4* to *JUnit5* (e.g., one package at a time), resulting in sparse *JUnit5* adoption. Typically migrated annotations are related to fixtures (i.e., @Before to @BeforeEach), and sometimes from @RunWith to @ExtendWith, or from @Category to @Tags.	19
JUnit3 to *JUnit4* migration	Automated migration from *JUnit3* to *JUnit4* using tool support.	1
TestNG to *JUnit5*	Remove TestNG in favour of *JUnit5* due to its popularity.	1
Mocking		**17**
Namespace error in mocking	Mocking frameworks, such as PowerMock, utilize an independent classloader, which may cause namespace error (i.e., class not found).	11
Mock usage	Adding Powermock to mock final, utility and abstract class.	5
Mock *vs* Injection	Removing mocking and use dependency injection instead.	1
Custom		**11**
Retry on failure/exception	Implementing custom annotations to retry test on failure (*JUnit4*). Developers should leverage *JUnit5* @RepeatedTest	8
Experimental	Implement custom annotation to categorize tests during feature addition to detect untested code, instead of using *JUnit4* @Category.	3
Mixed Framework Usage		**3**
Using @DependsOn in *TestNG*	Using *TestNG* in addition to *JUnit*, because *TestNG* provides the @DependsOn annotation, which enforces test ordering. However, developers can leverage *JUnit4* @FixMethodOrder to avoid using multiple frameworks.	2
Using @Test(group=X) in *TestNG*	Using *testNG* in addition to *JUnit*, because *TestNG* provides the group option inside @Test annotation, which categorizes tests. However, developers can leverage *JUnit4* @Category or *JUnit5* @Tag to avoid using multiple frameworks.	1
By-product	Changing/adding/removing test annotation due to feature deletion or test code relocation.	**10**

@Test(Timeout=X) (14%). Our analysis reveals innovative uses of timeout and maintenance problems of such uses due to software's ever-evolving nature. We find that timeout is employed to achieve various goals, i.e., detecting deadlocks (e.g., *Hbase-2428c5f*) and performance regressions (e.g., *Hadoop- f131dba8*), and providing meaningful debugging information when accessing external resources. While timeout is an effective tactic to serve the aforementioned purposes, we observe that developers constantly need to increase the timeout threshold (even by removing the use of timeout entirely) to avoid test failures and to accommodate the evolving software development. For example, once the running environment changes (e.g., to a slower cluster or platform), test execution may become slower and lead to timeout errors. Developers need to increase the timeout threshold to avoid test failures. Another example is that detecting performance regressions may become flaky as code evolves, i.e., the execution time comes closer to the timeout threshold and leads to unstable test results in different runs. To avoid flakiness, developers may increase the timeout threshold. This shows that timeout may not be suitable for performance testing, and a framework (e.g., OpenJDK JMH) that allows more sophisticated settings (e.g., repeated runs, warm-up iterations) should be leveraged.

> Developers use @Test(timeout=X) to detect concurrency issues or performance regressions; however, they may also relax/remove the timeout when performance regressions occur.

Fixtures (9.6%). Our fixture change analysis reveals the inherent difficulties and error-proneness in maintaining the balance between a clean test environment and minimized test execution time. To minimize test execution time, developers may continuously refactor test initialization code, i.e., extracting duplicate initialization code in a separate method with fixture annotations, or changing a fixture method from @Before to @BeforeClass (*Druid-da32e1ae*). However, some fixture code (e.g., resetting shared variables), if incorrectly placed in a class fixture (i.e., @BeforeClass), may violate the test independence assumption (S. Zhang et al., 2014), introduce bugs in test code, and produce unstable test results (i.e., flaky test). Therefore, developers may perform changes to execute such a fixture

code for each test method repeatedly (e.g., *Ambari-7153112e*). Interestingly, we noticed that developers expressed performance concerns on such changes and sometimes, they were even uncertain about whether such changes would completely fix the buggy test. Hence, developers may benefit from having a detection tool that helps them determine the trade-off (e.g., a search-based approach). Moreover, we observe that developers need to perform workarounds due to the limitations of expressing fixtures in JUnit. In particular, developers may remove JUnit fixture to resort to a direct call to the parameterized helper methods, increasing redundancies. This happens when there is a need to adopt different fixtures for each test method. However, JUnit does not support tailoring fixture, i.e., all the test cases in one test class share the same fixture. Another limitation is the lack of fine-tuning JUnit fixtures based on the test cases. For example, in *UpdateActiveRepoVersionOnStartupTest - Ambari* (2700bd125f), developers replaced JUnit @Before with a parameterized helper method to conditionally configure the cluster in the fixture based on test cases. We find that developers performed such workarounds due to JUnit limitations complicate test fixture and may increase test maintenance overhead.

> Developers are concerned about the tradeoff between minimizing test code duplication and minimizing test execution time. Furthermore, the lack of configuration capabilities in JUnit fixtures causes developers to perform workarounds, increasing technical debt and maintenance overhead.

@Category (10%). We find that developers mostly add @Category to categorize tests for test prioritization. In *Hbase*, the category groups test based on timeout thresholds. Developers acknowledge that categorizing based on timeout can improve regression testing practice by running faster tests first (i.e., the ones with smaller timeout threshold) to detect bugs quickly. We also find that developers add @Category to complement ignored tests for better maintainability. For instance, a "FailingTest" category can indicate ignored tests due to test failure. Developers may also use @Category to specify whether certain tests are integration tests or unit tests.

Developers customize @Category to assist diverse maintenance needs. We find that categorization based on test execution time can be useful for efficient regression test analysis, and can also help detect failed or ignored tests for more reliable maintainability.

@Parameterized (8.5%). We find that most changes to @Parameterized align to its regular use, i.e., increase the flexibility of managing test inputs. Developers may add more corner cases to the input using @Parameterized after adding new features or fixing bugs to avoid regression. We also find that developers may refactor the test code using @Parameterized. We find cases where developers use @Parameterized to reuse common test input arguments in different test cases. For example, in a test from *Hadoop* (ad1b988a8286), developers use @Parameterized to remove multiple subclasses that share the same code with only differences in test input. We also find that developers may use @Parameterized to run tests with different inputs in parallel to speed up test execution. However, one limitation of @Parameterized is that it can only be applied at the class level. Therefore, if only a subset of the test methods needs the parameterized annotation, there may be additional setup overhead that slows down the test. Finally, the JUnit4 Parameterized class contains an optional string pattern that helps decorate test results with additional messages for improved debugging. Including such messages is considered the best practice by some developers, as we found in our manual classification. Nevertheless, most of our manual classification samples do not leverage this best practice, and thus we believe users of JUnit4 can benefit from knowing about it.

@Parameterized is used to enable test code to take arguments and refactor test code and test input duplications. Moreover, developers can improve the test execution speed of a @Parameterized test by using parallel programming.

Expected Exception (4.9%). Developers sometimes need to test if the code would throw a specified exception for erroneous behaviours. Over the JUnit history, encoding expected exceptions in test cases can be achieved differently, i.e., *try* with *fail* in JUnit3, @Rule and @ExpectedException in JUnit4, and *assertThrows* in JUnit5. On the one hand, we find that

new JUnit releases gradually adapt to the increasing need for handling expected exceptions elegantly, i.e., a specialized annotation @ExpectedException instead of the workaround *try* with *fail*, and an improved annotation *assertThrows* to overcome the limitation of @ExpectedException. On the other hand, we find that developers may not be fully aware of new features in their adopted JUnit version.

For example, we find cases that developers may migrate back to JUnit3 to use the *try* with *fail* mechanism, because @ExpectedException is limited in providing comprehensive error messages. Such a backward migration is not needed, since developers could leverage @Rule from JUnit4, or migrate to JUnit5 `assertThrows`. In other cases, we find that developers tried to customize their test code to handle expected exceptions, but later migrated to a more framework-dependent pattern as described above. *Our findings suggest a potentially ill-defined knowledge of how to handle expected exceptions in practice. Developers could benefit from having an expected exception recommendation tool to reduce test maintenance overhead.* We also find that developers may use expected exceptions to facilitate test-driven development. Namely, the expected exception avoids test failures while developers actively work on implementing the features. Finally, we find a misuse of the expected exception associated with the *Exception* type. Developer discussions reveal that the use of general excepted exception types (e.g., `java.lang.Exception`) could hide the actual faults because the test will still pass if an unexpected sub-type exception is thrown. Therefore, one way to solve this issue is to use specific exception types instead of a general exception type.

> There is a diverse way to handle expected exception in test code. Developers sometimes are unaware of which mechanism to utilize and what mechanisms may be available in their adopted JUnit version.

@Rule (3.8%). JUnit4 introduced @Rule to provide a flexible mechanism to enhance tests by running code around a test execution, similar to @Before and @After. In alignment with the regular use of the @Rule, we find that developers often refactor duplicate test code using @Rule to improve maintainability and readability. We also found three common uses of built-in @Rules, namely the Timeout, TemporaryFolder, and TestName rules (JUnit

team, 2018). In these cases, developers preferred using these built-in @Rules to simplify test code. Future studies may consider using various annotations to help refactor test code.

> Developers utilize @Rule to remove duplicate code in test fixtures. Moreover, we find that developers can benefit from using built-in @Rules to simplify their test code.

@FixMethodOrder (1.4%). Prior studies (Eck et al., 2019; Lam, Godefroid, et al., 2019b; Lam, Oei, Shi, Marinov, & Xie, 2019; Luo et al., 2014a) found that one of the root causes of flaky tests is test order dependencies. JUnit4 provides @FixMethodOrder to allow a deterministic test execution order. For example, we find in *Camel* (4e7ec8f79b6) that since JDK7 does not preserve test execution order, developers fixed test flakiness using @FixMethodOrder. Hence, there is a future research opportunity on automated test fixing by applying such annotations.

> Developers apply @FixMethodOrder to ensure a deterministic test execution order and avoid dependency-related flaky tests.

Other Types of Test Annotations Changes

Migration (5.8%). In the migration from JUnit3 to JUnit4, we find that developers use an automated tool that applies migration in the entire codebase. However, the migration from JUnit4 to JUnit5 is applied manually and slowly (e.g., one package at a time). There are many migrations that started over one year ago (i.e., before 2019), but the issue reports still remain open today. We believe migrations to JUnit5 are intentionally manual, because some annotations, such as @Rule, are removed in JUnit5. Moreover, some annotations are renamed and may further cause confusion to developers (e.g., @Before is renamed to @BeforeEach). Therefore, developers could benefit from having an automatic JUnit5 migration tool. Finally, we find some changes where developers migrate from TestNG to JUnit5 due to the popularity of JUnit.

We find that the migration from JUnit4 to JUnit5 is done manually and slowly. To help developers utilize the new features in JUnit5, future studies should further investigate migration patterns for JUnit5 in order to assist developers with automated migration.

Mocking (4.7%). A JUnit class annotated with @RunWith indicates that the JUnit framework invokes a specified class using a developer-specified test runner instead of running the default runner. An issue with using mocking runners is that these mocking frameworks utilize an independent class loader, which sometimes causes namespace error due to conflicting classes. To resolve the issue, developers add @PowerMockIgnore to defer loading the conflicting classes. Finally, for one instance, we find that developers decided to remove mocking and use dependency injection instead.

Developers often use mocking for external dependencies. However, they may encounter issues related to namespace conflicts.

Custom Annotation (3%) and Mixed Framework Usage (0.8%). We found cases where developers created customized annotations to repeat the test execution upon failure (e.g., for detecting flaky test), or to indicate that a test is experimental. However, JUnit5 provides a new annotation @RepeatedTest, and both JUnit4 and JUnit5 provide annotations (e.g., @Category) to categorize tests. We also find cases where developers added annotations from TestNG, even though developers were already using JUnit, which provides similar annotations. The findings may indicate that sometimes developers might not know the functionality that is provided by testing frameworks, and may use customized annotations and increase maintenance costs.

Developers resort to customized annotations due to the lack of awareness about the features offered by testing frameworks, suggesting the need for annotation recommendation tool support.

By-product (2.7%). We find that developers may modify test annotations due to a feature removal or test refactoring (e.g., relocate the annotation to another test file).

5.5 Threats to Validity

Internal Validity. Our findings depend on the accuracy of our tool to mine annotation changes from the commit history. We mitigate this threat by validating our tool thoroughly. The extension of RefactoringMiner 2.0 detects annotation changes with a 99.7% precision and 98.7% recall.

External Validity. We study systems that are all open source implemented in Java, so the result may not be generalizable to all systems. To minimize the threat, we follow a set of criteria to select systems that are popular on GitHub, large in scale, and actively maintained. The studied systems cover various domains and are frequently used in commercial settings.

Construct Validity. We conduct a manual classification to understand the reasons behind test annotation changes. Due to the large number of changes, we take a statistically significant sample. There may be bias or misidentification in our manual classification on characterizing test annotation changes. Thus, two authors independently examined all available software artifacts and discussed them until the agreement is made. We do not claim to find all usage and misusage patterns, and the limitations of test annotations. However, we show the existence of such patterns and identify further research opportunities. Thus, future work should survey developers based on recent annotation changes to gain additional insights.

5.6 Implication & Contribution

Based on our empirical findings, we present actionable implications and future work for three groups of audiences: 1) researchers, 2) application developers and testers, and 3) framework designers.

5.6.1 Discussion and Implication for Researchers

R1: Test annotation is an integral part of test design and implementation. Future studies on test maintenance and refactoring should consider the peculiarity of test annotations. As we find in RQ1, test annotation changes are frequent in

test maintenance, and developers often use test annotations to improve test maintenance. As we found in RQ3, developers use various test annotations, such as fixtures or @Rule, to remove duplication in test code. Based on our manual classification, such refactorings are commonly performed, but there is a lack of tool support. Therefore, future refactoring studies may want to design test code refactoring techniques that leverage such test annotations.

R2: Developers may use test annotation for ad-hoc fixes, which may affect test maintenance or even code quality. Future studies are needed to study such impact and provide research solutions. In RQ3, we find that developers may use test annotations for ad-hoc fixes (e.g., adding @Ignore to failing tests or increasing the timeout threshold in @Test(timeout=X) without finding the root cause). Although the fixes make the tests pass, the underlying issues remain unsolved, which may cause more severe issues in the future. Moreover, as we found in RQ2, there are much more additions of @Ignore than removals, which indicates that many tests get disabled without being re-enabled. Future studies are needed to study the prevalence of such technical debt in ad-hoc test fixes and their potential consequences.

R3: There are future research opportunities on detecting misuses of test annotations. Recently, test smell detection starts to receive interest from both the academia and industry due to its practicality (Junior et al., 2020a, 2020b; Peruma et al., 2019a; Qusef et al., 2019; Spadini et al., 2018; Tufano et al., 2016). However, most prior studies only consider test smells related to test code, yet as we found in RQ3, there exist many test annotation misuses. For example, we found that developers use suboptimal ways to handle expected exceptions or @Parameterized in tests (e.g., tests expecting generic exceptions or not recording error messages when using @Parameterized). Furthermore, there are also possible test annotation misuses related to fixtures (e.g., using @Before without considering its performance impact). Our study highlights and opens an avenue for future research to better detect test smells and improve test quality.

R4: Future research is needed to provide automated test grouping and better utilization of test annotations to reduce test execution overhead. To reduce test

execution overhead, prior studies have proposed various test selection (Gligoric, Eloussi, & Marinov, 2015a; Rothermel & Harrold, 1997; Shi, Yung, Gyori, & Marinov, 2015; L. Zhang, Marinov, Zhang, & Khurshid, 2011) and prioritization techniques (J. Chen et al., 2017; Li, Harman, & Hierons, 2007; Mei et al., 2012; Rothermel, Untch, Chu, & Harrold, 2001; Saha, Zhang, Khurshid, & Perry, 2015; Thomas, Hemmati, Hassan, & Blostein, 2014; Yoo, Harman, & Clark, 2013; L. Zhang, Hao, Zhang, Rothermel, & Mei, 2013). In RQ3, we find that developers use @Category to group and execute small tests first (e.g., run faster tests first to detect failures early). Future studies may consider integrating their techniques with test annotations for better research adoption. Developers may also use parallelization to speed up test execution (e.g., using @Parameterized with thread pools). However, we find that JUnit5 provides a new annotation, @Execution(ExecutionMode.CONCURRENT), which allows parallel test execution. Future study is needed to assist developers to automatically adopt such annotations and improve test execution time without causing concurrency issues or flaky tests.

5.6.2 Discussion and Implication for Application Developers and Testers

A1: Developers need better education about the capabilities of testing frameworks. As found in a prior study (Chen, Shang, Hassan, Nasser, & Flora, 2016), due to differences in background, some developers may not be familiar with specific frameworks. In RQ3, we also have similar observations with testing frameworks. Even though JUnit is the most commonly used framework in Java (Zerouali & Mens, 2017), we find that some developers do not fully utilize test annotations. For example, some developers were unsure which way they should use to handle expected exceptions, and some developers created custom test annotations, even though JUnit already provides the same functionality. A prior study found that there is often a champion who first adopts new features in a framework and helps the team with adoption (Parnin, Bird, & Murphy-Hill, 2013). We recommend that developers follow similar procedures and dedicate at least one team member to gain expertise in testing frameworks and help the development team utilize testing frameworks.

5.6.3 Discussion and Implication for Framework Designers

F1: Framework designers need to provide better flexibility in their APIs. In RQ3, we find some cases where developers need to find workarounds or adopt other testing frameworks to bypass some inflexibility in JUnit. For example, @Before is executed for every test case in the class, but developers may only want @Before to be executed for a subset of the test cases. In this case, developers removed JUnit fixture annotations and replaced them with a direct call to the helper method, increasing redundancies. In other cases, fine-tuning fixtures based on test cases also enforced developers to use a Parameterized helper method over JUnit @Before. Finally, we also uncovered some limitations with @Parameterized in JUnit4, although it can remove test code and test input duplication and improve test coverage. Firstly, the JUnit4 parameterized class only works at class level, and cannot be configured at method level. Thus, for the specified test inputs, the test runner will execute every single test case even if not all test cases utilize the input. Hence, we believe that software engineering researchers may work with framework developers and identify possible issues that framework users encounter and improve the framework accordingly.

F2: Better annotation support targeting specific testing issues (e.g., flaky tests). We find that JUnit provides annotations, such as @FixMethodOrder, to resolve issues related to order dependent flaky tests. Similarly, one of the customizations we find is retrying on test failure. However, JUnit5 now provides a new annotation @RepeatedTest to help developers detect test flakiness more easily. To this end, with the recent research advances in the detection of flaky tests, we believe that incorporating more support in a practical framework, such as JUnit, will help developers quickly address test flakiness without resorting to other specialized tools that are difficult to adopt in practice.

5.7 Chapter Summary

This paper presents the very first empirical study on annotation changes in Java tests to fill the knowledge gap regarding the evolution and maintenance of tests, since prior studies focused mainly on the test code and ignored the test annotations. Our study reveals many

interesting findings with actionable implications:

(1) Test annotation changes are more common than test refactorings. Despite that, there is very limited tool support for migrating test annotations to newer framework versions or different frameworks, and automating common annotation change patterns within the same framework version.

(2) Test developers are sometimes *unaware of the features provided by testing frameworks*, and thus apply alternative suboptimal solutions. There is great need for tool support to detect the misuse (or lack of use) of annotations and recommend appropriate test annotations.

(3) Test developers are forced to apply workarounds to overcome the current *limitations of testing frameworks*. Framework designers need to be aware of these workarounds to improve the design and flexibility of test annotations.

Chapter 6

Demystifying test disabling practices to improve test maintainability

Software testing is an essential software quality assurance practice. Testing helps expose faults earlier, allowing developers to repair the code and reduce future maintenance costs. However, repairing (i.e., making failing tests pass) may not always be done immediately. Bugs may require multiple rounds of repairs and even remain unfixed due to the difficulty of bug-fixing tasks. To help test maintenance, along with code comments, the majority of testing frameworks (e.g., JUnit and TestNG) have also introduced annotations such as `@Ignore` to disable failing tests temporarily. Although disabling tests may help alleviate maintenance difficulties, they may also introduce technical debt. With the faster release of applications in modern software development, disabling tests may become the salvation for many developers to meet project deliverables. In the end, disabled tests may become outdated and a source of technical debt, harming long-term maintenance. Despite its harmful implications, there is little empirical research evidence on the prevalence, evolution, and maintenance of disabling tests in practice. To fill this gap, we perform the first empirical study on test disabling practice. We develop a tool to mine 122K commits and detect

3,111 changes that disable tests from 15 open-source Java systems. Our main findings are: (1) Test disabling changes are 19% more common than regular test refactorings, such as renames and type changes. (2) Our life-cycle analysis shows that 41% of disabled tests are never brought back to evaluate software quality, and most disabled tests stay disabled for several years. (3) We unveil the motivations behind test disabling practice and the associated technical debt by manually studying evolutions of 349 unique disabled tests, achieving a 95% confidence level and a 5% confidence interval. Finally, we present some actionable implications for researchers and developers.

Earlier version of this chapter was published in the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021). 12 pages. (D. J. Kim, Yang, Yang, & Chen, 2021)

6.1 Introduction

Modern software development handles an increasing complexity of feature enhancements. To ensure that the software quality remains on par with consumer expectations, software testing has been playing a pivotal role in software development. With the availability of JUnit and other testing frameworks, writing test code is becoming widely popular (D. J. Kim, Tsantalis, et al., 2021; Zerouali & Mens, 2017). Most large-scale systems utilize testing practices routinely and expose faults early.

However, test code can be subject to age and quality issues like the production code under test. For example, a test can also contain test-specific design issues that may hinder its ability to guard against regressions. Prior studies (Lam, Muslu, Sajnani, & Thummalapenta, 2020; Luo, Hariri, Eloussi, & Marinov, 2014c) have found that flaky tests may hinder the reliability of testing results that may fail for reasons other than recent changes. Similarly, researchers introduced the concept of test smells, which are design issues specific to the test code that may negatively impact test code comprehension and maintenance (Bavota, Qusef, Oliveto, Lucia, & Binkley, 2012b; Van Deursen et al., 2001).

To mitigate the efforts to fix broken tests, developers need to maintain and improve test code continuously. Challenges in maintaining and improving test code are more than fixing flaky tests and refactoring test smells. A prior study by Pinto, Sinha, and Orso (2012b) highlights both the importance and potential of studying the evolution of how tests are modified, added, or deleted. Understanding such evolution is essential to understand how the test becomes obsolete, why it is difficult to fix, and how it should be repaired. The findings can inspire future research and provide better testing support and tools.

In the context of software testing, developers can disable tests by commenting out the test method or class. In addition, with the introduction of annotations in Java 5, frameworks such as JUnit and TestNG introduce the annotations @Ignore and @Test(enabled= false), allowing developers to disable failing tests temporarily. Although disabling tests can be seen as an added flexibility for developers to alleviate maintenance difficulties, one can suspect that it may introduce technical debt.

With such flexibility, developers may disable hard-to-fix tests as a compromising solution to meet project deliverables. Despite the potential challenges that arise later from disabling tests, there exist limited studies on disabled tests as a source of technical debt. We believe that studying why developers disable test code is of paramount importance for both practitioners and researchers: (1) it indicates a source of potential technical debt that can direct future research efforts, (2) it may provide additional evidence on how bugs are fixed to improve automatic tool support, and (3) can help prevent future encounters of bugs.

To the best of our knowledge, there are no studies in the literature investigating disabled tests as a source of technical debt and providing tools for tracking all types of disabling changes in the test code. To address this issue, we develop an automated tool to identify all kinds of disabling and re-enabling practice at commit level: (1) a commented-out test code instance, (2) commenting out or deleting @Test, (3) using @Ignore from JUnit, and (4) setting @Test(enable=false) in TestNG. Our approach could detect the disabling/re-enabling practices with an overall precision of 96%.

In total, we study test disabling practices in 15 open-source systems of different sizes and from diverse domains. Our study focuses on understanding the disabled tests from the

following aspects: how often do developers disable a test (i.e., the prevalence and evolution) and why developers disable a test (i.e., motivations behind disabling and re-enabling a test). In particular, we answer the following three research questions (RQs):

RQ1: How common are test disabling changes? Test disabling practices are 19% more common than regular test refactorings, such as renames and type changes. Even though we find that disabling tests are prevalent, there is no prior study on how they may affect test maintenance.

RQ2: What is the change pattern of disabled tests? Through analyzing the change pattern and final destination of the disabled tests, we find that most disabled tests stay disabled. Many of the disabled tests have been disabled for several years. We also find that for the disabled tests that are resolved, many are deleted directly.

RQ3: Why do developers utilize test disabling practice? We conduct a manual classification to uncover test disabling practices and how they are used to bypass maintainability challenges. We find that most tests were disabled in the first place due to issues such as test failures, but many tests remain disabled even when the bugs are fixed. Some bugs may be marked as "Won't Fix" with the tests being disabled. We also find that developers often use disabling changes to handle other maintenance challenges in testing, such as test dependency and refactoring. Our findings highlight potential future directions on helping developers improve test maintenance and detect potential issues in test code.

In summary, our findings provide actionable implications for two groups of audiences:
(1) **Researchers**: We open an avenue for further research directions on detecting disabled tests and their relation with other aspects of software development (i.e., quality, maintainability, and performance improvements). We also highlight potential directions on assisting developers in tracking disabled tests and their co-evolution with production code.
(2) **Developers**: Our findings reveal the usage of disabled tests in many different aspects of test maintenance. Disabled tests may be used in ad-hoc ways to hide real faults or bypass test failures. Most tests are temporarily disabled until a fix is found; however, the disabled tests are not re-enabled after the fix (i.e., a bug report is closed). These findings indicate the need to assist developers with the best testing practice to trace disabled tests in practice.

6.2 Method for Detecting Test Disabling Practice

In this section, we discuss our methodology for tracking test code evolution to identify test disabling-related changes. Tracking the evolution of program elements in commit history is an ongoing research direction, and most existing tools are not designed specifically to work on annotations and commented-out code (Grund, Chowdhury, Bradley, Hall, & Holmes, 2021; Pham & Yang, 2020b; Tsantalis et al., 2020b; Tsantalis, Mansouri, Eshkevari, Mazinanian, & Dig, 2018b). Therefore, we propose an approach that detects disabled tests through analyzing annotations and comments and tracks the evolution of the detected disabled tests. Our approach first detects all tests, including both disabled and active (i.e., enabled) tests in each studied version (Section 6.2.2), and second performs version-by-version comparison to track the evolution of the tests (Section 6.2.3).

6.2.1 Definition of Test Disabling Practice

When using testing frameworks such as JUnit or TestNG, developers can use the @Test annotation to specify a method or methods in a class as test cases. As systems evolve, some tests may become obsolete or require some changes, and developers may want to temporarily disable a test. Developers can use framework-supported annotations (i.e., @Ignore), removing the @Test annotation, or commenting out the test method/class to disable a certain test. Our goal is to study such test disabling practices in the codebase and their potential impact on test maintenance. In particular, we consider the following code changes as test disabling changes (in contrast, we consider the reverse operations as test re-enabling changes):

(1) Adding @Ignore at both the class and method level.

(2) Setting @Test(enable=false).

(3) Deleting @Test annotation.

(4) Commenting out the entire test method or class.

6.2.2 Detecting Disabled and Active Tests

Given one version, our approach first detects all the disabled and active (i.e., enabled) tests. Detecting active tests is straightforward: We leverages JavaParser to find all the tests with an *@Test* annotation as the testing frameworks (JUnit 4 and 5, and TestNG) of the studied systems require to have such an annotation for tests.

Detecting disabled tests requires one to design techniques per each type of test disabling practice. Each type of disabling-related change (as defined in Section 6.2.1) corresponds to one unique type of disabled test. The first two types, i.e., adding `@Ignore` and setting parameters in `@Test`, will result in adding explicit annotations to the disabled test methods. The third type will produce methods in test classes without an `@Test(...)` annotation. The fourth type will result in complete test methods embedded in comments. Similar to detecting active tests based on `@Test` annotation, for the first three types, we also utilize JavaParser for analyzing method annotations. For the last type, we propose an algorithm to identify commented-out test methods.

Analyzing Annotations for Detecting Disabled and Active Tests

For detecting active tests and some types of disabled tests, we use the off-the-shelf JavaParser (Tomassetti, Smith, Maximilien, & Kirsch, 2021) to extract annotations per method in test classes. The following rules are applied to decide the status of a method in test classes.

- If the annotations contain `@Test` (without enable=false parameter), the associated method represents an active test.
- If the annotations contain either `@Test(enable=false)` or `@Ignore`, the associated method represents a disabled test.
- If the annotations do not contain `@Test`, the associated method represents a *candidate* disabled test.

Note that the lack of `@Test` annotation is indefinite to decide a disabled test since it

is common for developers to write non-test methods (e.g., helper methods) in test classes. Such *candidate* disabled tests will be further confirmed through analyzing the evolution (i.e., tracking). If a previous/later version of a candidate disabled test is an active test, this test method is confirmed disabled for the current version.

Detecting Disabled Tests in Comments

In addition to the disabled tests expressed by annotations, we also identify the tests that are disabled through commenting out. Algorithm 1 describes how we extract disabled tests in comments. The input *commentTarget* is either a block comment or a list of comments in consecutive lines. A *commentTarget* may contain zero or more commented-out tests. For one *commentTarget*, our tool first detects an `@Test` annotation (line 8) and then continues to detect a '{' in the following comment lines. The text in between could be an annotation and a method signature. Then we use a JavaParser API *parseMethodDeclaration* to confirm whether the text is indeed a method signature. In addition, we use the existence of a paired right bracket to filter out incomplete test methods that only contain method signatures.

Algorithm 1 Commented out test method detection

```
 1: Input: commentTarget
 2: Output: coTests, i.e., is a list of commented-out tests
 3: function detectCOTests(commentTarget)
 4:     lines ← commentTarget.split( "\n" )
 5:     coTests ← []
 6:     for i←0; i¡lines.length; i++ do
 7:         line←lines[i]
 8:         if line contains @Test then
 9:             location ← location of the first occurence of '{'
10:                     in this or the following lines
11:             break if location is null
                                   ▷ notMethodSignature is based on a JavaParser API
12:             continue if notMethodSignature(
                          strInBetween( i, location ) + "{}")
13:             end ← findEndOfMethodBody( lines, location)
14:             cotests.add( ¡the detected commented-out test¿ );
15:             i←end
16:         end if
17:     end for
18: end function
```

6.2.3 Tracking the Evolution of Disabled Tests

Upon detecting (candidate) disabled and re-enabled tests in one version, we continue to track the evolution in the commit history, and pinpoint the related changes of such tests, e.g., one disabled test is later re-enabled. Tracking the evolution requires our approach to match program elements in every two consecutive versions.

We adapt RefactoringMiner to perform the matching because 1) RefactoringMiner is shown to have the highest precision (96.6%) and recall (94%) compared to other refactoring tools and AST diff tools (Tsantalis et al., 2018b); and 2) RefactoringMiner can detect various types of refactoring operations, such as method/class renaming, moving and extracting methods, and modifying method signatures. Furthermore, we incorporate tracking commented-out tests in our approach. As commented-out code may be incompatible with the live code and may result in compilation errors (not supported by RefactoringMiner), we use a lightweight approach to handle the commented-out test code. In particular, a *TEST ID* (*<fully qualified class name>::<method name>(<method parameter types list>)*) is constructed based on the extracted information of commented-out tests (Section 6.2.2). For each commented-out test method, its *TEST ID* is compared with the ones in the parent and child commits for finding the paired test methods. The abovementioned points allow our approach to collect an accurate and comprehensive dataset for our study.

For each pair of matched tests in two consecutive versions, if the status of the test is modified from active (i.e., enabled) to disabled, we decide the commit is disabling-related changes. The reverse status change is determined as a test re-enabling change. If there is no matching test in the latter commit, the unmatched test is deemed deleted. If there is no matching test in the prior commit, the unmatched test is deemed newly added by the current analyzed commit, e.g., one commit may introduce commented-out tests.

Approach Evaluation. We performed an evaluation of our approach with regards to 1) detecting disabled tests in comments (Section 6.2.2) and 2) detecting disabled tests in commit history as these two steps may produce incorrect results.

For 1) detecting disabled tests in comments, we took all the 168 detected commented-out

Table 6.1: Precision of test tracking, i.e., detecting the three types of commit changes, including disabling a test, re-enabling a test, and deleting a disabled test.

	Disabling a test	Re-enabling a test	Deleting an disabled test	Total
Sample	364	122	98	584
Precision	98%	96%	92%	97%

tests, examined them manually to decide the correctness. Our approach yields a precision of 100%. However, we cannot evaluate the recall due to the lack of oracles. For 2), as tracking is performed at commit level (e.g., whether one commit is disabling-related), we used stratified sampling to take a statistically significant (95%±5%) sample of 584 cases on three types of changes, namely disabling a test (364), re-enabling a test (122), and deleting a disabled test (98). Then we manually examined the correctness of each sampled commit. Table 6.1 shows the precision of the three types of changes. Our approach achieves a precision of 97% for all the sampled cases and a precision of 98%, 96%, and 92% for the three change types, respectively.

Upon manual examination, we identify the following sources of false positives. First, due to framework migrations, developers may need to add or delete annotations. For example, migrating from TestNG to JUnit4 will remove the `@Test` on the class, which will be detected as ignoring a test class, and adding `@Test` on each method will be detected as unignoring test methods. Second, due to the limitations of RefactoringMiner, duplicating one file to two similar files and merging two files into a single file cannot be detected. For example, in Apache Camel (*9da3f5af*), the *Web3jConsumerIntegrationTest* is duplicated to *Web3jConsumerTransactionsTest* and *Web3jConsum-erLogTest*. However, RefactoringMiner only reports *Web3jConsumer-IntegrationTest* is renamed to *Web3jConsumerTransactionsTest*. Thus, we detect *Web3jConsumerLogTest* as a newly added class. In Apache Flink (*8d3a74f9*), *StatefulJobSavepointFrom12MigrationITCase* and *StatefulJobSavepointFrom13MigrationITCase* are merged to *StatefulJobSavepointMigrationITCase*, but RefactoringMiner only reports *StatefulJobSavepointFrom12MigrationITCase* is renamed to *StatefulJobSavepointMigrationIT-Case*. Thus, we detect *StatefulJobSave-pointFrom13MigrationITCase* as a deleted class.

Table 6.2: An overview of the studied systems (from 2015 to 2020).

Systems	Test LOC 2015	Test LOC 2020	Source LOC 2015	Source LOC 2020	Total # Commits
Camel	355K	533K	289K	607M	22,584
Cassandra	28K	78K	177K	216K	4,336
Cloudstack	51K	80K	1M	519K	4,698
Druid	22K	147K	64K	242K	4,181
Flink	82K	371K	105K	407K	13,997
Hadoop	349K	660K	463K	810K	13,668
Hbase	168K	287K	433K	409K	7,271
Hive	104K	269K	524K	1M	8,221
Ignite	162K	500K	253K	577K	15,397
Incubator-pinot	1.6K	75K	11K	196K	6,179
Kafka	2K	133K	11K	132K	5,845
Maven	14K	17K	44K	48K	660
Openfire	1.2K	5.2K	179K	94K	2,097
Orientdb	74K	188K	140K	360K	8,452
Storm	2.7K	35K	61K	240K	4,554
Total	1.4M	3.3M	3.8M	5.9M	122K

Lastly, RefactoringMiner does not work for commented-out tests, so renaming a commented-out test will be detected as deleting a commented-out test and adding a new commented-out test.

6.2.4 Studied Systems

Table 6.2 shows an overview of the studied systems. To obtain high-quality repositories to make our results more reliable, we select the studied systems by following three selection criteria. First, we selected the top 1,000 Java systems on GitHub ordered by popularity (i.e., stargazer count). We also ensured that the repositories are not forks as they may not be part of the main branch and not actively maintained. Although it would be interesting to study disabled tests from other branches that consist of feature additions or bug fixing activities, as they may involve more frequent usages of test disabling, our research only considers disabled tests that are merged into the main branch, as they may indicate more challenging maintainability tasks that could not be resolved at the time. Second, we discarded the systems that are below the 90th percent quantile in terms of size (i.e., lines of

code), repository popularity (i.e., stars), and the number of commits collectively. Namely, we only study repositories that fall inside the top 10% in all of the mentioned criteria. Finally, we discarded inactive repositories that did not have any commits in 2020. We ended up with 15 systems, i.e., Camel, Cassandra, Cloudstack, Druid, Flink, Hadoop, Hbase, Hive, Ignite, Incubator-Pinot, Kafka, Maven, Openfire, Orientdb, Storm. We analyze the code changes in these systems from January 2015 to January 2020. These studied systems cover different domains, ranging from distributed databases, stream processing frameworks, message brokers, and group chat servers.

6.3 Results

6.3.1 RQ1: How Common are Test Disabling Changes?

Many prior studies focus on studying maintenance challenges caused by technical debt (Cunningham, 1993; Liu et al., 2018; Pham & Yang, 2020b; Tan, Yuan, Krishna, & Zhou, 2007). However, there is less empirical evidence on the technical debt in the test code thus far, especially on technical debt related to disabling practices. Developers may disable tests as code evolves, which may cause future maintenance challenges. As a stepping stone to understanding test maintenance challenges, in this RQ, we study the frequency of test disabling practices.

Approach. We study how frequently developers disable a test at both class and method levels. Disabling tests at the class level would prevent executing all test cases within the class, whereas disabling at the method level would stop executing a single test case (e.g., a method with an `@Test` annotation). To provide a comparative statistic, we show the prevalence of test disabling changes (i.e., disable/re-enable/delete) along with common test code transformations at the same program element level by following prior studies (Ketkar, Tsantalis, & Dig, 2020a; D. J. Kim, Tsantalis, et al., 2021). Specifically, we compare test disabling changes at method level with Rename Method, Rename Parameter, and Change Parameter Type, and at class level with Rename Class. We use a tool, called RefactoringMiner, implemented by Tsantalis et al. (2020b) to detect rename and type changes.

Table 6.3: Frequency comparison between test disabling changes and the common test code refactorings at the same program element level.

	Method Level		Class Level		Total	Total
	Total	Per Commit	Total	Per Commit	Changes	Commits
Test Disabling Changes	2,581	2.7	530	0.6	3,111	949
Refactoring	2,495	1.9	120	0.1	2,615	1,334
Δ % Percentage	+3.4%	+42%	+341%	+500%	+19%	-29%

Table 6.4: The frequency of various types of test disabling-related changes. *Disabling Tests* shows the total number of changes that disable tests. *Re-enabled Tests* shows the total number of changes that re-enable tests and whether developers simultaneously modified the tests (i.e., modified vs. unmodified). *Deleting Disabled Tests* shows the number of changes that delete disabled tests.

	Disabling Tests	Re-enabling Tests		Deleting Disabled Tests
		Modified	Unmodified	
Method	2,581	314	486	762
Class	530	115	96	87
Total	3,111 (62.5%)	429 (8.6%)	582 (11.7%)	849 (17.1%)

Tsantalis et al. (2020b) reported that RefactoringMiner could detect refactoring activities with an average precision of over 99% and recall of over 93%. Despite differences in the two practices, such comparison is reasonable because these common code refactorings occur at the same program level.

Result. ***Test disabling changes is prevalent during test maintenance and has a similar change frequency compared to test refactorings.*** Table 6.3 compares the prevalence of the disabling changes with that of the refactoring changes in test code. As shown in Table 6.3, at the method level, the number of test-disabling changes is comparable to the number of test refactorings (i.e., +3.4% difference), and at the class level, the test-disabling changes are performed more frequently than the rename class refactorings (i.e., +341% differences). We also observe that the total test-disabling changing commits are less than the total test refactoring commits. Despite this, at both method and class level, the average test-disabling changes per commit are higher than that of test refactorings (i.e., +19% difference). Based on these results, we find that test disabling changes are prevalent in practice and are comparable to traditional refactorings.

Table 6.4 presents the frequency of three types of test disabling-related changes: disabling a test, re-enabling a test, or deleting a disabled test. We find that 62.5% (i.e., 3,111/4,971) of the disabling-related changes disable the test, which is the most frequent change among all the change types. 20.3% (i.e., 1,011/4,971) of the disabling-related changes re-enable the test. Moreover, a non-trivial percentage (i.e., 42%) of the re-enabling changes modify the test. In other words, many of the disabled tests remain the same when they are re-enabled. We also find that a significant number (17.1%) of the test disabling changes delete disabled tests. In the next RQ, we further study the destination of each disabled test and its change pattern.

Developers frequently disable tests in software development, and the frequency of such practice is comparable to common refactorings at the same program element level. We also find that many disabled tests may be re-enabled without any changes or may be deleted directly.

6.3.2 RQ2: What is the Change Pattern of Disabled Tests?

As shown in RQ1, test disabling practice has a non-negligible presence during test maintenance and evolution. In this RQ, we further study the evolution patterns of disabled tests, their destination (e.g., finally re-enabled or stay disabled), and how long a test remains disabled. Studying the evolution of disabled tests may quantitatively show the process of how developers maintain disabled tests, whether the corresponding issues are fixed immediately or persist for a long time, and whether the corresponding issues are improperly fixed and cause the tests to become disabled again in the future.

Approach. Our goal is to study the life-cycle of every disabled test. In RQ1, we analyze the test disabling changes by studying their frequency. However, bugs that are hard to fix may cause multiple rounds of changes to the test code. Hence, we carefully track the evolution history by utilizing the full commit-level history of the disabled test to report their evolution pattern and final destination. To study the change pattern more accurately,

Table 6.5: Change patterns of disabled tests. Unresolved tests represent the tests that remain disabled at the end of the studied period. Resolved tests represent the tests the are either re-enabled or deleted completely at the end of the studied period.

Change Pattern	Frequency
Change patterns for unresolved tests	
DISABLED	1,229
DISABLED → RE-ENABLED → DISABLED	21
DISABLED → DELETED → DISABLED	1
Total	*1,251*
Change patterns for resolved tests	
DISABLED → RE-ENABLED	871
DISABLED → DELETED	824
DISABLED → RE-ENABLED → DISABLED → RE-ENABLED	46
DISABLED → RE-ENABLED → DISABLED → DELETED	24
DISABLED → RE-ENABLED → DISABLED → RE-ENABLED → DISABLED → RE-ENABLED	1
Total	*1,766*

we trace refactoring activities (e.g., rename) performed on disabled tests, using RefactoringMiner (Tsantalis et al., 2020b). Additionally, we study the longevity of disabled tests by mining the number of days between the changes that disable the tests and the changes that either re-enable or delete the disabled tests.

__Result.__ ***Many disabled tests (41%) remain disabled in the studied period. For the resolved disabled tests, 47% were deleted in the codebase.*** Table 6.5 shows the evolution patterns of disabled tests, which highlights how disabled tests evolve ever since they were born. Note that since a test may undergo multiple rounds of changes in the studied period, the total frequency of evolution patterns should be lower than RQ1 where we report the total raw frequency. We also categorize the statistics into unresolved and resolved cases to indicate the final destination of disabled tests. Unresolved tests refer to the tests that stay disabled, and resolved tests refer to the ones either being re-enabled or deleted. We find that most disabled tests (41%, 1,251/3,017) stay unresolved as the destination. For the resolved cases, 28% (848/3,017) become deleted, and 30% (918/3,017) become re-enabled. There are 21 cases where developers tried to resolve a disabled test but eventually changed them back to being disabled. After some investigation, we find that these tests are disabled again due to three main reasons. First, developers revert the commit due to a mistake. Second, developers re-disable the test code later when the test

Table 6.6: The distributions of the average time (in days) for a disabled test to become re-enabled, deleted, or remain disabled (i.e., developers did not modify the test in the studied period after it was re-enabled).

	Time (in days)					
	Min.	25%	50%	75%	Max.	Mean
Re-enabled	8.4	17.1	39.4	65.9	431.2	61.8
Deleted	4.8	19.9	92.3	222.1	696.6	158.7
Remain Disabled	142.0	508.0	793.2	1096.4	1475.7	797.4

fails again. Third, developers re-disable the test code whenever there is a version update of one external dependency. There are also 24 cases where developers tried to resolve the ignored test but eventually deleted them. We observe that these tests were disabled for a long time and may have become obsolete as developers suggest deleting the tests instead of updating them.

Table 6.6 shows the distributions of the average time (in days) it takes for a disabled test to become re-enabled or deleted (i.e., resolved). We also show similar statistics for the tests that stay disabled (i.e., unresolved). We observe that the median time for developers to re-enable a test is 39 days. However, it often takes over three months (median is 92 days) for developers to delete a disabled test. In general, there is a higher possibility that a disabled test may become deleted if it has been disabled for a more extended period. One likely explanation is that many of the disabled tests may have become obsolete. We notice similar patterns of obsoleteness across all types of disabled tests.

Finally, for the tests that remain disabled, most of them have been disabled for several years. It is likely that these disabled tests are "forgotten" by developers and remain in the codebase.

In RQ3, we manually study the reasons for the tests to be disabled and re-enabled.

Overall, it takes a longer time for the disabled tests to be deleted (median time is three months) than to be re-enabled (median time is 39 days). Many tests that remain disabled have been disabled for years.

6.3.3 RQ3: Why do developers utilize test disabling practice?

As previous RQs reveal, disabling tests is a ubiquitous practice during software evolution. The test disabling practice is a double-edged sword. On the one hand, it provides developers with convenience in bypassing test-related issues. On the other hand, it may be used to bypass some maintenance difficulties, which can result in a silent and long waiting period for the tests to be re-enabled, if ever. Test disabling mechanism may hinder software reliability as the disabled tests may remain disabled indefinitely in codebases. In particular, there is a lack of tools to manage the life cycle of disabled tests and assist developers in proactively re-enabling the temporarily disabled tests. In this RQ, we perform a manual classification to understand why developers utilize test disabling practice, i.e., the scenarios that developers utilize such convenience of disabled tests. Categorizing the scenarios will reveal the common challenges developers may face in maintaining disabled tests and test maintenance in general. Obtaining such understanding on disabling tests will inspire future tools that can better manage disabled tests for improving quality assurance activities.

Approach. To understand the motivations of utilizing test disabling practice, we analyzed and categorized a statistically significant sample of disabled test instances. We combined the disabled tests from all the studied systems and adopted the stratified sampling technique to sample each studied system independently for producing a statistically significant sample using 95% confidence level and a 5% confidence interval. We use a 95% confidence level because it provides a high level of certainty that the true population parameter falls within the specified range. We also found 9 incorrectly detected instances by our tool, i.e., a 2.6% false positive rate, and excluded them in Table 6.7.

We manually study and understand the reasons that developers disable tests by analyzing all available software artefacts: including issue ID from the Jira bug report, GitHub's pull request/issue tracking, commit messages and the code changes. Some commit messages contained sufficient information to understand the reason behind test disabling practices.

Our manual classification involves two phases:

Phase I. The first two authors of the paper (A1 and A2) independently derived an initial classification by manually inspecting the relevant software artifacts such as commit messages, test code, comments surrounding the test code, and bug reports if available. Additionally, we use *git log* to check out other relevant commits on the same set of modified source code files to gain supplementary insights if the current commit lacks sufficient information.

Phase II. A1 and A2 unified the derived reasons and compared the assigned reason for each evolution pattern. Any disagreements were discussed until a consensus was reached. We used Cohen's Kappa inter-rater agreement to measure the degree of agreement between the two authors (Cohen, 1960). Cohen's Kappa considers a scenario where the agreement between two authors is purely by chance. In our manual classification, the inter-rater agreement of the coding process had a Cohen's kappa of 0.7, indicating a substantial level of agreement (Cohen, 1960). To encourage the replication of our results, we have made the dataset available[1].

Result. Table 6.7 shows our manually derived taxonomy of the reasons developers disable the tests. Below, we discuss each category in detail.

Hiding Test Failure (40%). Developers frequently disable some tests when test failures occur. The most common cause (81/131) is that bugs are introduced during software maintenance. While working on fixing the bugs, developers may temporarily disable the failing test cases, especially when the bugs require non-trivial effort and time to fix. However, only 32/131 of the disabled tests are re-enabled after the relevant bugs are fixed. In this case, we also notice that developers do pay a certain effort to provide some traceability of the disabled tests, such as creating bug reports on disabled test cases as a reminder to re-enable such tests. In contrast, some issues are difficult to fix, and test failures may have persisted for a non-trivial time. Developers adopt test disable to remove test failures explicitly without working on fixing the bugs. *7/131 of the tests were disabled due to failure and were never re-enabled in our studied period.* For such cases, test disabling practice is used by developers as a convenient way of bypassing test failures while keeping the test code in the

[1]https://github.com/boyang9602/FSE_Ignore_Test

Table 6.7: manual classification result: a taxonomy of why developers disable tests.

Categories	Motivation	#Frequency
- Hiding Test Failures		**133 (40.6%)**
Disabling tests while working on bug fixes	Developers temporarily disable failing tests while working on fixing the bugs.	81
Flaky test	Developers may disable flaky tests to avoid occasional failures.	42
Won't-fix bugs in code/test	Developers disable tests to avoid failures as they will not work on fixing the failures due to difficulty.	7
Slow test	Test failures due to long running time, so developers disable such tests to avoid failures.	2
Library incompatibility	JUnit annotation (@Parameterized) was not supported by Ant in the used CI platform. Thus, the test was disabled.	1
- Precautions During Feature Maintenance		**78 (23.8%)**
New/Improved Features	In the process of implementing new or improving existing features, developers may temporarily disable relevant tests or introduce new tests that are disabled (e.g., commented-out tests) to avoid potential failures.	62
Deprecation	Developers may disable relevant tests in the process of deprecating features.	12
Refactoring	Developers may disable tests during refactoring.	4
- Diverting to Manual Testing		**38 (11.6%)**
Require manual input	Developers need to manually run tests that need to be manually configured (e.g., database setup and secret keys).	24
Expensive Test	Developers need to manually run tests that are expensive to run and may not have to be run all the time (e.g., performance and migration test).	13
Experimentation	Developers manually run tests that are under experimentation.	1
- Dependency Issue		**21 (6.4%)**
Difficulties in maintaining external dependencies	External dependency is hard to maintain due to various reasons. Developers may need to disable tests when there are bugs in the external dependency, unexpected version changes, or dependencies are hard to integrate or use.	17
Waiting for functionality update in external dependency	Developers disable tests while waiting for feature improvement in the external dependencies.	4
- Test Design Issues		**16 (4.9%)**
Selective test inheritance	Developers disable tests to disable unneeded inherited tests, while selectively reusing some inherited tests.	14
Redundant Test	Developers disable tests that are redundant and covered by a different test class.	2
- Other Reasons		**41 (12.5%)**
Unknown	Lacks of explicit mention of why tests are disabled (e.g., no relevant Jira issues and comments).	32
Obsoleteness	Developers disable obsolete tests.	5
disable by mistake	Developers accidentally disable the test.	4

repository. However, we found that there is no traceability provided for developers to track these disabled tests. We could not find any mention of the disabled tests in Jira issues or in documents. These disabled tests may remain forgotten with the issues remain unfixed.

Another main motivation (42/131) is to avoid test failures caused by flaky tests. As flaky test results are nondeterministic, diagnosis can be challenging. After developers fix the issues (i.e., flaky tests no longer fail), they may re-enable the tests. In our studied cases, *only 7/42 flaky tests are re-enabled later*, aligning with the study by (Lam et al., 2020), who reported that over half of flaky tests remain unfixed.

We also uncovered two test failures due to slow tests, for which developers simply disable the slow tests without either fixing the test or the source code. Finally, one failure is caused by library incompatibility in Travis CI. In *Openfire (ddb20ffe)*, developers discuss that @Parameterized is not supported by the Ant build system installed in Travis CI, causing failure in tests where @Parameterized is used. Developers disable the affected tests as a convenient solution while waiting for the incompatibility to be resolved by framework developers.

In total, only 31% of the tests are re-enabled after the bug fix. The remaining disabled tests are either disabled indefinitely (i.e., due to lack of solution) or deleted. We also find that, for such permanently disabled tests, there is often a lack of traceability in GitHub or Jira issues.

Precautions during Feature Maintenance (23%). Developers commonly utilize test disable practice to avoid potential test failures during maintenance activities, such as adding new features and refactoring. For 62/78 cases, we find that developers precariously disable the test cases that may fail in the process of feature implementation due to incomplete functionalities (e.g., the feature takes more than one commit to finish). For example, in *Flink (df448625)*, developers commented out the tests related to retrieving log files and left a comment saying that *"TODO activate this test after logging retrieval has been added to the new web frontend"*. We also notice that developers may indicate relevant bug issue IDs when disabling test cases. For example, in *Hadoop (18fe65d75)*, developers left a comment as *"requires HDDS-801. Requires once feature is in place"*. Although we find that developers

may try to add traceability on disabled tests, the current practice remains ad-hoc (i.e., by using only comments in the code). Developers may use disabled tests to harbor some temporary tests and delete such tests after the feature is complete, i.e., being replaced by an official test (e.g., *Orientdb (0fc9bee1). Nevertheless, among the 62 cases we examined in this category, only 30 tests are re-enabled, and the others remain disabled in the studied systems.*

In addition to new features, we find that developers may disable the tests of deprecating features (instead of removing them). Developers may choose to disable the tests temporarily while replacing deprecated features and adopt the disabled tests once the new implementation is finished. However, we observe there are cases where the tests for the deprecated features remain disabled in the codebase. Lastly, developers sometimes disable tests during refactoring (4/78). For example, in *Openfire-97f7cf3f* and in TrustStoreConfigTest.java, developers commented out the entire class prior to extracting OpenfireX509ExtendedTrustManagerTest and removing few code duplications using a helper class, KeystoreTestUtils.java.

> Developers may temporarily disable tests during feature maintenance or refactoring. However, we find that 50% of such disabled tests remain disabled.

Diverting to Manual Testing (11.2%). Developers may disable some tests with the plan to manually running them. For the studied cases in this category, test reconfigurability was the most common problem causing developers to disable the test (24/38). For example, we find that some tests are disabled for manual testing due to the need to manually configure the access key and secure key (e.g., *Camel (ba22a8175f94a)*) or setup databases (e.g., *CloudStack (96c38bf4)*). Such cases require manual testing as the tests depend on resources that must be manually started or configured prior to the test execution. Another 13/38 of the tests in this category are disabled because they are expensive to run (e.g., migration or performance tests). For example, as discussed in *Flink (b7ae3e5338): ManualWindowSpeedITCase*), *"When doing a release, we should manually run theses tests on the version that is to be released and on an older version to see if there are performance regressions."* However, it is

difficult to know if developers remember to manually run these tests before each release, and disabling these tests may result in higher maintenance costs (e.g., only discovering issues before the release).

Finally, we find one rare case in *Camel (bd1661b248): JavaSocketTests*, where the developer introduces a commented-out test code as it is part of experimental code.

> Developers manually test expensive resources that should only be executed under particular circumstances, such as migration or performance testing.

Dependency Issue (6.2%). Without good mocking strategies, we find that external dependencies may become hard to maintain (17/21) and a source of technical debt, causing tests to fail and be disabled. For example, in *Camel (35b83b1d)*, we find that bugs in the external dependencies cause test failures (i.e., *"Upgrade smack due to bug in smack 4.0.6"*). In another case, in *Camel (40ae73c4)*, due to unexpected version changes in external dependencies, the test fails and is disabled (i.e., *"It looks like the problem is embedded XMPP server. It was overridden to 2.21 currently"*).

In total, only 5/17 cases were re-enabled after the issue was fixed. Interestingly, as shown in *IGNITE-9920*, the test continues to be disabled even if the bug report is reported closed with a *WONT'T FIX* resolution due to difficulties in test maintenance. The other tests remain disabled due to similar reasons. Developers may also disable tests while waiting for new features to be added in external dependencies. For example, in *Druid (4b3bd8bd)*, developers comment out several tests as they need to wait for an external dependency (i.e., Joda) to release a new feature. However, we find that only 1/4 cases become re-enabled later.

> Developers may disable tests due to issues/updates in external dependencies. However, 71% of the disabled tests in this category remain disabled or are deleted due to reasons such as test maintenance difficulties.

Test Design Issues (5%). Developers may disable tests when changing/improving test design. We find that developers may leverage inheritance to reuse part of the tests in parent

classes. However, developers may disable some of the inherited yet unneeded tests in child classes. A reverse may also happen where developers re-enabled the disabled tests in the child classes (*Ignite (63b9e1653d)*). Our finding shows that there may be maintenance or designing challenges when handling test inheritance. Therefore, developers need to decide whether an inherited test should be executed or not. Additionally, there are cases where developers find the same tests that exist in multiple test classes and decide to disable the redundant tests.

> Developers may disable tests to bypass test design limitations related to test inheritance.

Other Reasons (12.5%). We categorize the remaining disabled tests that do not belong to any of the above-mentioned categories as *"Other Reasons"*. We find that there are 32 cases of disabled tests where we cannot find any discussion/comment. We categorize these cases as unknown. For example, developers may only include a commit message, such as *"Disable Test"*, without any other explanation or reference to other software artifacts (e.g., Jira), where only 7 out of 32 were eventually re-enabled.

We find 5 cases where developers disable the tests because they become obsolete. Finally, there are 4 cases where developers disable the test accidentally (e.g., commented out the test) and thus were re-enabled immediately afterward.

> Developers did not provide the reasons nor traceability (e.g., Jira issues) for many (32/338) of the disabled tests. Most of such disabled tests remain disabled in the studied systems. We also find cases where developers disabled obsolete tests, and they may also disable some tests by mistake.

6.4 Threats to Validty

In this section, we discuss threats to validity of our study.

Internal Validity. Since commented code may exist in countless different formats, it may be impossible to find a generalized rule to detect all the cases. Table 6.8 shows one such example, where there is a mix between the commented-out code and natural language.

Table 6.8: An example of an invalid block comment that cannot be detected by the tool. Invalid comment represents a mix between natural language and code.

Original Method	Commented using Block Comment

<pre>1 @Test
2   /* this is a demo*/
3   public void demo() {
4 }</pre>
	<pre>1 /*@Test
	2 this is a demo
	3 public void demo() {
	4 }*/</pre>

Therefore, our tool may not be able to detect all the commented-out methods. Nevertheless, to minimize the threat, we treat the consecutive line comments as a single target and detect each target as multiple commented-out methods. We apply the same rule when there could be blank lines between the different parts of a commented-out method. Namely, we allow our rules to be tolerant for a single line of code, but we treat it as two different targets for a number higher than one. Despite this issue, the precision of our tool for test status is high (98%), where in fact there are no false positives in terms of commented out tests. Our approach also relies on RefactoringMiner to detect and track refactoring changes. As shown by Tsantalis et al. (2020b), RefactoringMiner has high precision and recall, which should not affect our results. It is possible that some commented-out code is only meant as a template code for future implementation and could be non-compilable. Our tool does not check for these instances, and they may exist in our study and skew our understanding of the manual classification. Moreover, we only consider @Test annotation to determine the test method and test class. It is also possible that files that use an older version of Junit (i.e., Junit3) may not use @Test annotation to indicate a test method or class. Our tool does not check for these instances and may miss some disabled tests. Finally, we do not consider partially commented out tests, such as commenting out assertions. Compared to disabling a test completely, partially commented out tests may involve more complex change, such as when tests contain multiple assertions in a test.

External Validity. Our studied systems are all open source implemented in Java, so the result may not be generalized to all systems. To minimize the threat, we follow a set of

criteria to select systems that are popular on GitHub, large in scale, and actively maintained. The studied systems cover various domains and are frequently used in commercial settings. Future studies are encouraged to replicate our experiment on other systems and systems implemented in different programming languages.

Construct Validity. In RQ3, we conduct a manual classification on the reasons that the tests become disabled. We conduct the study on a statistically significant sample using a 95% confidence level and 5% confidence interval. To reduce the biases in our manual classification result, two of the authors independently studied the sample and compared the results. Any discrepancy is discussed until a consensus is reached. We computed the Cohen's Kappa, and found that the level of agreement is substantial between the two authors (0.7).

6.5 Discussion and Implication

Based on our empirical findings, we present actionable implications and future work for two groups of audiences: 1) researchers and 2) application developers and testers.

6.5.1 Discussion and Implication for Researchers

R1: Developers use disabled tests to bypass test failures, which may affect test maintenance and code quality. Future studies should investigate its impact on software quality. As we find in RQ2, tests may be hard to fix immediately and may remain disabled for an extended period. Some re-enabled tests are again disabled later due to inadequacy of the bug fixes as bug fixing tasks are difficult. Moreover, as found in RQ3, developers may disable tests to temporarily hide test failures, such as disabling flaky tests, while the bugs remain unfixed. Although these tests are necessary for revealing faults, they may no longer guard the software against regressions and uncover the possible presence of new bugs. Therefore, future research must study the impact of disabled tests on software quality from two aspects. First, an interesting direction is to study the relationship between disabling tests and software defect proneness to provide additional insights into the impact of disabled tests on software quality. Second, future studies may quantitatively assess

the impact of disabled tests on fault detection capabilities using mutation analysis (Jia & Harman, 2011).

R2: There is a lack of automated support on tracking disabled tests, which may lead to "forgotten" tests. Future studies may consider providing traceability support to developers. During our manual classification, we notice that many disabled tests are not referenced in issue reports and are not tracked by any software artifact. For example, developers may commit test disabling changes with only mentioning in the commit message that a test is disabled. In some cases, we find that the bug may be resolved, but the test is still disabled. Due to the lack of traceability and documentation, most of these disabled tests may then be "*forgotten*", and stayed disabled for several years and are never re-enabled. Future research may consider providing automated traceability to track the disabled tests and better assist developers during test evolution.

R3: Developers may disable some tests and divert them to manual testing. Future studies should investigate approaches to provide better automation support. As we find in RQ3, 11% of the tests are disabled because they are difficult to run automatically or are time-consuming. However, delaying test execution may result in accumulated maintenance overhead (e.g., bugs are only revealed at the late stage of the development or before the release). Since these tests need to be manually executed, it is also possible that developers may forget to run them. Future studies may also investigate better mocking approaches and adopt test reduction or prioritization (Gligoric, Eloussi, & Marinov, 2015b; Peng, Chen, & Yang, 2020a) to ensure these tests can still be included in part of the continuous integration process while maintaining acceptable test overhead.

R4: Future studies should investigate the impact of test obsolescence and the co-evolution between test and source code. In our study, we find that developers use test disabling for a wide range of maintainability tasks, yet only a few were ever re-enabled in practice. Many of the disabled tests are deleted as they become obsolete. As systems evolve, some tests may become outdated and may need to be updated Marsavina, Romano, and Zaidman (2014); Zaidman, Van Rompaey, Demeyer, and Van Deursen (2008); Zaidman, Van Rompaey, Van Deursen, and Demeyer (2011). However, it is not clear to which

degree do developers maintain tests to keep up with the development, and whether there are replacement tests for the tests that were deleted. Future studies on test obsolescence and their co-evolution with source code may provide better support to developers on test maintenance.

6.5.2 Discussion and Implication for Developers

D1: Assisting developers with best testing practices about how to maintain disabled tests. As found in RQ1 and RQ2, test disabling is prevalent in test maintenance, but most disabled tests remain disabled. Developers often disable tests due to a bug; however, they rarely open a new Jira issue to track the process of re-enabling the test code. For example, out of the 141 samples that remain disabled (RQ3), only 10% are related to unresolved issues and 22% of the tests that remain disabled do not have any documentation on Jira. For the remaining 68% of the disabled tests, they remain disabled even after the issues were fixed. As an example, in *Hive (22df53b6)*, several tests remain disabled and forgotten even after their bug issues were closed (e.g., HIVE-18341). To better trace these disabled tests, developers should consider adding Jira issues, or similar artifacts, to document tests that require an update. Otherwise, the non-traceability of disabled test code may lead to worse software quality in the long term.

6.6 Chapter Summary

Similar to source code, there are bugs and maintenance challenges in test code. As a result, developers may bypass a test failure by disabling the test, i.e., adding `@Ignore` or comment out the test. Such tests disabling practices, when misused, may cause technical debt and harm the long-term maintenance. In this paper, we conduct the very first empirical study on test disabling practice in Java systems. We first implement a tool to detect and track the changes of disabled tests in software development history. Then, we conduct both quantitative and manual classification on test disabling changes. We find that: 1) Test disabling is a prevalent practice in test evolution and has a similar frequency level compared

to test refactoring at the same program element level. 2) Most disabled tests remain disabled and have been disabled for years. Many of the disabled tests are either re-enabled without any code change or deleted directly. 3) Our manual classification highlights the reasons for the tests to be disabled. We find that most tests are disabled due to maintenance challenges (e.g., flaky tests and required to do manual testing) rather than waiting for bug fixes. Moreover, most disabled tests remain disabled even after the bugs are fixed. Our discussions provide possible future research directions to further improve test maintenance and suggestions to developers to better track disabled tests so that they are not "forgotten".

Chapter 7

Demystifying inheritance in test code and impact on test redundancies

Inheritance, a fundamental aspect of object-oriented design, has been leveraged to enhance code reuse and facilitate efficient software development. However, alongside its benefits, inheritance can introduce tight coupling and complex relationships between classes, posing challenges for software maintenance. Although there are many studies on inheritance in source code, there are limited studies on using inheritance in test code. In this paper, we take the first step by studying inheritance in test code, with a focus on redundant test executions caused by inherited test cases. We empirically study the prevalence of test inheritance and its characteristics. We also propose a hybrid approach that combines static and dynamic analysis to identify and locate inheritance-induced redundant test cases. Our findings reveal that (1) inheritance is widely utilized in the test code, (2) inheritance-induced redundant test executions are prevalent, accounting for 13% of all execution test cases, and (3) the redundancies slow down test execution by an average of 14%. Our study highlights the need for careful refactoring decisions to minimize redundant test cases and identifies the need for further research on test code quality.

Earlier version of this chapter was published in the the 46th International Conference on Software Engineering (ICSE-2024). 12 pages. (D. J. Kim, Chen, & Yang, 2024)

7.1 Introduction

In a highly evolving software market, customers expect new features delivered on time alongside reliable and high-quality products (Abrahamsson, Salo, Ronkainen, & Warsta, 2017). To reduce maintenance cost and improve productivity, code reuse plays a pivotal role. Through code reuse, developers can take the advantage of existing functionality and achieve faster development while maintaining code quality. Particularly, one of the main advantage of inheritance, a fundamental aspect of object-oriented design, is to facilitate code reuse (Oracle, 2022a, 2022b). Inheritance offers a simple way for a Class A to reuse a feature defined in Class B by utilize the `Extends` keyword, such as Class A extends Class B.

While inheritance provides many benefits in reducing implementation and maintenance overhead, using inheritance ineffectively can also create tight coupling between classes (stackoverflow, 2013b), causing non-flexibility and overly redundant code (stackoverflow, 2020). Many researchers found that ineffective use of inheritance is correlated to software quality issues and maintenance difficulties Marinescu and Codoban (2014); Nasseri, Counsell, and Shepperd (2008); Prechelt, Unger, Philippsen, and Tichy (2003); Tahir, Counsell, and MacDonell (2016). Prior studies even used inheritance as a proxy to measure software complexity and to predict software defects in industry systems (Chidamber & Kemerer, 1994; Chowdhury & Zulkernine, 2011; Subramanyam & Krishnan, 2003; Zimmermann, Nagappan, & Williams, 2010).

Existing studies on inheritance primarily focused on the source code (Chidamber & Kemerer, 1994; Chowdhury & Zulkernine, 2011; Marinescu & Codoban, 2014; Nasseri et al., 2008; Prechelt et al., 2003; Subramanyam & Krishnan, 2003; Tahir et al., 2016; Zimmermann et al., 2010). However, there has been limited investigation into the impact of

inheritance in the test code; especially inherited test cases. Our preliminary analysis reveals that 40% of 503 sampled open-source software systems use inheritance in test classes, indicating a significant adoption of inheritance in software testing. One potential benefit of test inheritance, as found by X. Wang, Xiao, Yu, Woepse, and Wong (2021), is that developers often turn to inheritance to mock the source code under test. Another benefit of inheritance is test code reusability, which can improve coverage and help test maintenance (Biddle & Tempero, 1996).

Despite the potential benefits of test case inheritance, using inheritance in test code can also lead to overly complicated code as software systems become more complex. A study by Peng, Chen, and Yang (2020b) showed that test case inheritance causes most code dependencies, which can over-complicate test case design and maintenance. Moreover, some practitioners view inheritance as poor practice and should be refactored (stackoverflow, 2013a), i.e., "*Prefer composition over inheritance and interfaces*" (stackoverflow, 2020) or "*It is a bad idea to use inheritance in test*" (apache dev, 2023). In addition to test case design, one issue with test inheritance is that it can result in multiple *subclasses* inheriting identical test cases from the same *superclass*. Such inherited and identical test cases are redundant and can cause test execution overhead, which further extends the already time-consuming testing process.

In this paper, we aim to study the impact of inheritance in the test code by focusing on the redundant test executions caused by inherited test cases. We develop a hybrid approach that combines static and dynamic analysis to study and detect inheritance-induced redundant test executions. First, we apply static analysis to analyze the source code and extract the inheritance hierarchy in test classes. Then, we extract test cases candidates that potentially cause redundancies. Finally, we apply dynamic analysis, which involves source code coverage and test oracle analysis, to detect whether these candidates are truly redundant. We conduct our study on 15 open-source Java systems and found that (1) 13% of the total executable test cases are in fact redundant, and (2) such redundant tests take 14% of total test execution time. Finally, we shed light on (3) the challenges of addressing redundancy, precisely the difficulty in preserving code coverage while removing redundancy.

This paper makes the following contributions below:

- We are one of the first studies that provide evidence on the prevalence of inheritance usage in test code: over 40% of the analyzed systems use inheritance in test.
- We propose a hybrid approach that combines static and dynamic analysis for pinpointing inheritance-induced redundant test execution, providing developers with insights to identify bottlenecks in test case executions.
- We find that a non-negligible number of test cases are introduced through test case inheritance, accounting for 13% of the total executable test cases and 14% of the total test execution time.
- We find that eliminating redundancy poses challenges in preserving code coverage, as inherited test cases can be redundant in certain *subclass* but non-redundant in the rest.
- We release the source code of our tool and the dataset[1] of our experiments to help other researchers replicate and extend our study.

7.2 Motivation

Existing studies on inheritance have primarily focused on the source code (Chidamber & Kemerer, 1994; Chowdhury & Zulkernine, 2011; Marinescu & Codoban, 2014; Nasseri et al., 2008; Prechelt et al., 2003; Subramanyam & Krishnan, 2003; Tahir et al., 2016; Zimmermann et al., 2010). However, there has been limited investigation into the impact of inheritance in the test code. We conjecture that developers regularly use test code inheritance in practice. To verify whether developers use inheritance in test code, we analyze the number of test code inheritance that is attributed to various software systems, by mining hundreds of open-source Java Repositories, similar to technique the employed by (Kang, Yoon, & Yoo, 2023). We start with the Java-med dataset from Alon et al. (Alon, Brody, Levy, & Yahav, 2018), which consists of 1,000 top-starred Java systems from GitHub. We utilize *Spoon*, a static analysis tool, to create the source/test code model for the entire repository (Pawlak,

[1] https://anonymous.4open.science/r/ICSE2024-D32B/README.md

Table 7.1: Analyzed repository characteristics.

Repository Inheritance Characteristics	**#Repositories**
Has at least one inheritance tree in the test code	202
No inheritance tree in the test code	301

Monperrus, Petitprez, Noguera, & Seinturier, 2015). From the list of `Java` files in the repository, we check (i) whether a file is a `test` file and (ii) whether the `test` file is part of the inheritance tree. To measure the prevalence of inheritance, we choose to use the inheritance tree rather than simply counting inheritance usage (e.g., *Extends*). The inheritance tree offers a more comprehensive view, representing the hierarchical structures of test classes and the holistic relationships among classes in the repository.

To check if a file is a `test`, we examine if the name's *Prefix* or *Suffix* contains the *"T/test"* keyword. To determine whether a `test` file is a component of the inheritance tree, we search for `Extends` keywords and iteratively traverse upward through its `superclass` until we reach the root node of the inheritance tree. During traversal, a test class itself may be a `superclass` for other test classes. Hence, we also traverse through all the `subclasses` until reaching a leaf class. We omit systems that do not have test classes. Accordingly, from the 1000 repositories, we are left with 503 repositories as shown in Table 7.1. We find that the ratio of the repositories that have at least one inheritance hierarchy tree is 40% among 503 studied repositories. The finding suggests that a significant number of repositories in fact adopt inheritance in software testing. This percentage is a staggering proportion considering that many practitioners view inheritance as poor practice and should be refactored, i.e., *"Prefer composition over inheritance and interfaces"* (stackoverflow, 2013a) or *"It is a bad idea to use inheritance in test"* (apache dev, 2023; stackoverflow, 2020). More interestingly, there is a moderate to strong correlation (i.e., 0.61) between the number of test files in the repository and the number of test inheritance hierarchy (Akoglu, 2018). The finding indicates that as the software becomes complex, developers may be more inclined to use test inheritance. Based on this analysis, we believe that inheritance plays a significant role in the design of test code.

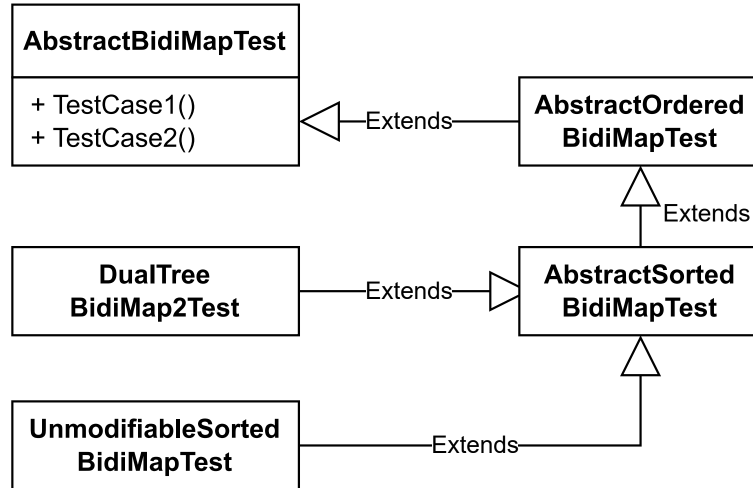

Figure 7.1: Developers re-use test through inheritance to ease maintenance.

Figure 7.1 shows a real-life scenario of test inheritance in *Commons-collections*, where the developer uses inheritance for test code reuse and easing maintenance. For example, test cases 1 and 2 from *AbstractBidiMapTest* cover the coverage of different source code functionality, *DualTreeBidiMap* and *UnmodifiableSortedBidiMap*. Despite the aforementioned advantages, the prevalence of inheritance in test code shows a potential challenge: the proliferation of redundancies within test cases. For example, as shown in Figure 7.1, not only is the inheritance deep, but the two test cases declared in the root class can also be inherited throughout the hierarchy. The impacted subclasses with the `abstract` modifier, i.e., *AbstractSortedBidiMapTest*, *AbstractOrderedBidiMapTest*, do not execute the test cases. However, the concrete *subclasses*, i.e., *DualTreeBidiMap2Test* and *UnmodifiableSortedBidiMapTest*, in the leaf position will eventually execute the test cases, as the testing frameworks (e.g., JUnit) will instantiate them during testing, which leads to potential redundant test case execution, i.e., proliferation. While such redundancies may have no consequences (e.g., performance overhead) in the source code, their presence in test code represents a considerably detrimental practice.

Unlike source code, where redundant code may remain dormant and unexecuted, test cases annotated with the `@Test` annotations are automatically executed within the inheriting child classes. As shown in Figure 7.1, while such a test case aims to help maintenance and

code reuse, it may also lead to redundancies if the coverage and test oracle remain the same. This fundamental distinction forms the basis of our research focus, which aims to identify redundant test cases lacking fault-locating capabilities while contributing to an increase in test execution time. Our investigation seeks to shed light on these intricacies of inheritance-induced redundancies in test code.

Based on the aforementioned discussion about redundant test cases, we focus on two important definitions: (1) ***Redundancy-Inducing Inheritance:*** Inheritance is considered redundancy inducing if it declares test cases and has impacting *subclasses*, and (2) ***Inheritance-Induced Redundant Test Candidates:*** Test cases are classified as potentially redundant if they are executed more than once due to inheritance, although they may or may not be truly redundant. In Figure 7.1, the example represents (1). The root class, *AbstractBidiMapTest*, contains test cases and has many impacting *subclasses*. Its test cases are hence (2), as they are executed multiple times in the *subclasses*. In the subsequent section, we present our technique for detecting ***Inheritance-Induced Redundant Test Executions***, given (1) and (2).

7.3 Related Works

Inheritance Evolution and Maintenance. Many works investigated the evolution of inheritance in source code. For example, Shaheen and du Bousquet (2008) studied the relationship between inheritance and the number of methods to test. They claim that testing should be more expensive if the inheritance depth is high, as the inherited method should be retested. Nasseri et al. (2008) studied whether inheritance evolves breadth-wise or depth-wise, and developers consider depth-wise as hard to maintain and prefer breadth-wise inheritance. Nasseri, Counsell, and Shepperd (2010) studied the evolution of inheritance from the perspective of class re-location to understand what motivates their move and try to give insights on potential maintenance challenges. Giordano et al. (2022) studied the evolution and impact of delegation and inheritance on code quality. They find that their evolution often leads to code smell severity being reduced and improved maintainability.

Inheritance Maintenance in Test Code. Limited works investigated the evolution and maintenance of inheritance in the test code. The work by X. Wang et al. (2021) conjectured that despite the existence of powerful mocking frameworks, developers often turn to inheritance to mock source code under test. Hence, they proposed a tool to refactor mocking via inheritance with a mocking framework. In contrast, our analysis shows the existence of inheritance in the test case design and its implication on test execution overhead. There is another body of work relevant to our work. Peng et al. (2020b) studied the impact of code dependencies on continuous integration. They found that inheritance causes the majority of dependency in test cases and proposed test dependency-related smells. While their work is the most relevant to our work, they emphasize test dependencies and little on the impact of test code redundancies.

Test Case Minimization/Reduction. Our work is related to research in test case reduction/minimization, which focuses on eliminating redundant test cases while preserving fault detection capability. Nadeem, Awais, et al. (2006) developed `TestFilter` that uses the statement-coverage criterion for the reduction of test cases. McMaster and Memon (2007) proposed a new metric called average expected probability to help minimized test cases that retain fault-detecting capability. Fang and Lam (2015) used assertion fingerprints to detect similar test cases that can be refactored into one single test case. Alipour, Shi, Gopinath, Marinov, and Groce (2016) presented an approach that reduces a test suite by compromising a certain amount of coverage while preserving the overall fault-finding ability. Other approaches focus on test case generation that involves test case reduction to remove redundancy in the generated tests. Nirpal and Kale (2011) proposed a Genetic Algorithm to generate test cases. Fraser and Arcuri (2012) proposed a novel technique implemented as a part of `EvoSuite` to generate test suites with high coverage. Their results show that smaller test suite gives higher coverage and adopted their fitness function to reduce redundant tests. These studies focus on minimizing redundant test cases by removing entire test cases. Our work, on the contrary, is more related to the design of the test code, which makes test case removal non-trivial. Complementary to test case reduction, other works focus entirely on reducing time execution of test executions (Groce, Alipour, Zhang, Chen,

& Regehr, 2014; Khalek & Khurshid, 2011; J. Kim, Kim, & Yoo, 2017). Bell and Kaiser (2014) used unit test virtualization to reduce the time required for the execution of a test suite. J. Kim et al. (2017) utilized general purpose GPU to improve the execution of genetic algorithms. Our work, on the contrary, focuses on identifying redundant test cases, which may also help reduce test execution time. Moreover, while these works show great promise in improving test case execution time, issues of redundancy still exist. There is work by Vahabzadeh, Stocco, and Mesbah (2018) that is most relevant to our work. They perform fine-grained test case minimization by merging all test cases that have the same code coverage. However, due to the complexity of inheritance relationships, they

7.4 Technique for Identifying Inheritance-Induced Redundant Test Executions

Due to the complexity of inheritance trees and the scale of modern software, developers may not always be aware of redundant test executions caused by inheritance. In this section, we present our technique to detect redundant test executions. Figure 7.2 summarize the overview of our technique, which consists of three main parts: static analysis to detect (1) ***Redundancy-Inducing Inheritance*** and (2) ***Inheritance-Induced Redundant Test Candidates***, and (3) dynamic analysis to identify ***Inheritance-Induced Redundant Test Executions***. Performing static analysis before dynamic analysis reduces the cost of the latter since not all test cases are *Inheritance-Induced Redundant Test Executions*. Therefore, we perform dynamic analysis on a subset of the total test cases, specifically on the ***Inheritance-Induced Redundant Test Candidates***.

7.4.1 Statically Detecting *Redundancy-Inducing Inheritance*

In this section, we describe how we extract redundancy-inducing test inheritance. For this task, we use *Spoon*, a static analysis tool (Pawlak, Monperrus, Petitprez, Noguera, & Seinturier, 2016).

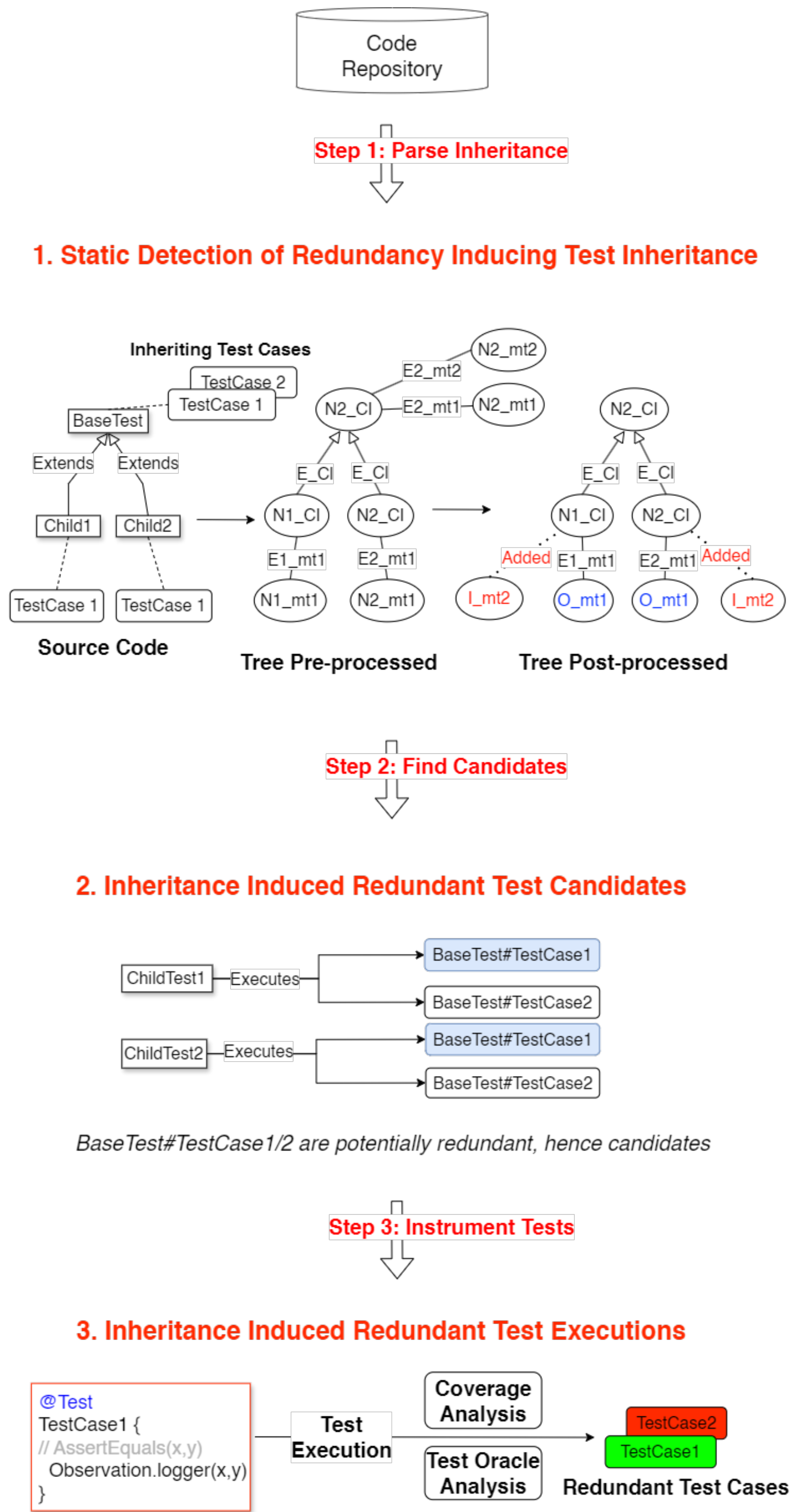

Figure 7.2: Overview of the End-to-End Process for Finding Redundant Test Cases.

Identifying Test Cases

In our detection of redundant test cases, our first step is to identify the relevant test classes from the *.java* files. Our studied systems use *JUnit4+* testing frameworks to design test cases and utilize *Maven* to execute the test cases. Hence, we first search for all potential test cases that exist within the studied systems. We look for methods that are annotated using the @Test annotation. However, not all test cases written in the test code are executed during regression testing. A test case can either be disabled to prevent flaky test or excluded in the *Maven* build, specified in the `pom.xml`, according to development needs (D. J. Kim, Yang, et al., 2021). Hence, we execute our test cases using *Maven* and filter out *skipped* test cases.

Extracting Inheritance Hierarchy

We then determine whether the identified test classes form an inheritance tree, i.e., `extends superclass`. For every test class, we use the *Vistor* Pattern to recursively visit its `superclass` until the terminating condition is met, such as Java's root Object() class or a class from external libraries. When traversing upstream, a test class itself may be a `superclass` for other test classes. Hence, we also traverse through all the `subclass` until reaching a `leaf` class. Once we have traversed all reachable test classes, we generate a comprehensive tree hierarchy denoted as $Tree$, containing crucial information. We represent nodes 1) N_cl as test classes, and nodes 2) N_mt as test cases. We represent edges 1) E_cl as a directed edge between two N_cl (e.g., node1 and node2), where the node1 is the subclass of node2, and 2) E_mt is the non-directed edge between N_cl and N_mt, where N_mt is a test case of test class, N_cl. As shown in the Figure 7.2, this step is described by transformation from *source code* to *tree-preprocessed*.

Identifying Inherited Test Cases

In Section 7.4.1, we assigned test case nodes, N_mt, to its corresponding test class, N_cl. Based on this *Tree*, we now extract the following test case types: unique methods,

overridden methods, and inherited methods, which are annotated as U_mt, O_mt, and I_mt. To elaborate, U_mt are methods that are unique to the class, O_mt are methods that override the parent method, and I_mt are methods inherited from its `superclass`. More formally, we consider a method to be O_mt, if and only if (1) they have the same signature, i.e., the same method name, the same number of parameters, and are not static, (2) the method is a subtype of a supertype method and (3) type erasure of the parameter is equal for generic types (Pawlak et al., 2015). Once O_mt is detected, identifying U_mt and I_mt becomes straightforward. Any method signature statically present in N_cl is classified as U_mt for that particular class, while any method inherited from the *superclass* is categorized as I_mt. Following the definitions (1-3), we generate a post-processed inheritance hierarchy, termed $Tree_post$, where test cases are accurately annotated. This transformed *Tree* is now ready for further analysis of redundant test cases. Refer to Figure 7.2 for a visual representation of this transformation from *tree-preprocessed* to *tree-postprocessed*.

7.4.2 Statically Detecting *Inheritance-Induced Redundant Test Candidates*

As discussed in Section 7.2, our focus is to detect redundant test case execution caused by inheritance. For this, we first perform static analysis to identify potential redundant candidates. In particular, we analyze the $Tree_post$ generated in prior Section 7.4.1 in the following ways: 1) We first determine the test classes, T_cl, that are `non-leaf` class in the $Tree_post$ and have executable test cases, T_mt. Such T_cl indicate potential sources of inherited test cases, 2) We then determine whether T_cl contain more than one `subclasses`, and if the resulting `subclasses` is a `non-abstract` class that can execute the inherited test case. Then, such test cases are executed more than once and are potentially redundant. More formally, if a test case T_mt1 is inherited by both `non-abstract` $subclassA$ and $sublassB$, then such T_mt1 has the potential of being a redundant candidate.

7.4.3 Detecting *Inheritance-Induced Redundant Test Executions* through Dynamic Analysis

Whether T_mt1 is a truly redundant test case cannot be fully determined by static analysis discussed in Section 7.4.2. For example, a $subclassA$ and $subclassB$ may both inherit T_mt1, but through a program dependency, `subclasses` can alter the state of the T_mt1 (e.g., by setting environment variables). In this case, T_mt1 can no longer be considered redundant as its behavior may be intentionally written by the developer to achieve partial code reuse by subclassing. However, it is challenging to statically determine the ground truth of program dependency to guarantee that T_mt1 is redundant. Hence, we resort to dynamic analysis by executing the test cases to collect execution trace information. Specifically, we instrument the test cases prior to execution to collect (i) source code coverage and (ii) test oracle information. Our intuition is that if the T_mt1 that is executed in both $subclassA$ and $subclassB$ has the same code coverage and test oracle, then T_mt1 is truly redundant. This technique draws inspiration from prior works in test case amplification (Danglot et al., 2019; Hossain, Dwyer, Elbaum, & Nguyen-Tuong, 2023a) and test case reduction (Di Nardo, Alshahwan, Briand, & Labiche, 2013; Konsaard & Ramingwong, 2015; McMaster & Memon, 2007; Nadeem et al., 2006; Vahabzadeh et al., 2018; Yoo & Harman, 2012), where code coverage and oracles are considered to validate test case quality. Below, we discuss the detail of our dynamic analysis.

Extracting Code Coverage

To detect redundant tests, we conduct code coverage analysis on *Inheritance-Induced Redundant Test Candidates*. We use *JaCoCo* to collect coverage data during test execution. Below, we outline the process for executing 1) inherited test cases and 2) instrumenting *JaCoCo*.

Executing Inherited Tests. In our analysis, we run test cases one-by-one to avoid accumulation of dirty states that may affect the test result, i.e., coverage and oracle. Hence, we

re-initialize the JVM and reset the environment for every test execution. However, executing test cases in *Maven* is non-trivial for inherited test cases. Such inherited test cases are not directly present in the test code, yet may be executed many times through inheritance (e.g., a test case in the `superclass` is inherited by multiple `subclass`). To execute the inherited test case, we use *Maven-Surefire*'s command-line options called `-DTest` with specified test cases such as `subclass#inherited_method`. Here, `subclass#inherited_method` refers to the test case inherited from its `superclass`, while `subclass` represents the class that inherits the test case. In addition, we observed that running a single test case in a multi-module project may only rely on a compilation of a specific subset of modules rather than on the entire project. Hence, we improve test execution time to speed up the experiment by leveraging the *-pl* option with a comma-separated list of modules to remove the compilation of unnecessary modules. However, we may miss some dependent modules when using the `-pl` option. Hence, we use the options `-am` to build all the dependent modules of the specified modules. For example, if *module_A* depends on *module_B*, using `-am` will build both modules. Finally, *Maven* may run numerous static analyses in the default build, such as *License check* and *CheckStyle*, which are not required to execute test cases. We also remove these to improve test execution time. Finally, we run these options in the root directory to successfully execute a single inherited test case in a multi-module project.

Collecting Code Coverage. We use *JaCoCo* to generate the code coverage report at three levels, i.e., instruction coverage, branch coverage, and line coverage. *JaCoCo* is one of the most popular code coverage tools that instruments bytecode to trace test execution (Jacoco, 2023). We integrate *JaCoCo* as a *Maven* plugin by configuring the *pom.xml* of the studied projects. While integration is simple for most *Maven* projects, for the multi-module *Maven* project, the coverage report is only limited to classes within the module, and will not be shown for test cases covering classes outside of the modules (e.g., integration test). Therefore, as some test cases cover multiple modules, we add an extra *report-aggregate* goal to the parent *Maven* build script (i.e., the main pom file).

Acquiring Test Oracle

In software testing, *Test Assertion* plays a critical role in assessing whether the actual behavior of the program aligns with the expected behavior specified by the developers (i.e., the test oracle) (Danglot et al., 2019; Gupta, Sharma, & Pachariya, 2019; Hossain et al., 2023a; Yoo & Harman, 2012). If the observed values of the program state differ from the oracle, the test assertion fails, indicating that the program is incorrect. Test failures indicate software regression caused by buggy code introduced through developer modification. Hence, in addition to coverage, the quality of assertions becomes crucial in assessing the effectiveness of test cases in capturing faults within the source code (Hossain, Dwyer, Elbaum, & Nguyen-Tuong, 2023b; Jia & Harman, 2009; Papadakis et al., 2019). Hence, to detect redundancies in test cases, we use both code coverage and test assertion, ensuring effective identification of redundant test executions.

We instrument the test assertions to collect the state of the program for both expected and actual behavior during test execution. Our test instrumentation is lightweight. We examine the static import of the studied systems to uncover all potential testing frameworks (e.g., JUnit/Hemcrest) that developers may use to write test cases. From these frameworks, we extract all the APIs used for assertion. During the test instrumentation, we identify such APIs and replace these APIs with `print` statements, which collect the program states during test execution. However, during our instrumentation, we uncovered that assertions are written in a variety of contexts in the test code. In particular, (1) The test cases may rely on reusable method which performs the oracle analysis. In this case, the test case itself may not have assertions. Hence, we leverage *Spoon* to instrument the entire codebase to collect more accurate test oracles.

Execution Time of our Technique

As discussed previously, our technique consists of three important steps: detecting 1) *Redundancy-Inducing Inheritance*, 2) *Inheritance-Induced Redundant Test Candidates*, and 3) *Inheritance-Induced Redundant Test Executions*. The execution time for steps 1) and 2)

Table 7.2: Systems Studied and Their Inheritance Statistics.

Project	#Inheritance tree	#Test classes constituted by inheritance tree	#Test classes within entire codebase
Commons-collections	10	206 (85%)	243
Zookeeper	6	301 (80%)	378
Avro	46	233 (49%)	478
Maven	13	83 (37%)	227
Shiro	10	45 (36%)	126
Commons-math	27	124 (31%)	400
Feign	6	30 (28%)	108
Iotdb	21	110 (25%)	437
Shenyu	7	133 (16%)	838
Dubbo	32	122 (14%)	856
Graphhopper	11	36 (14%)	252
Rocketmq	5	23 (11%)	214
Commons-lang	6	16 (10%)	161
Pdfbox	3	17 (8%)	213
Biojava	3	11 (4%)	259
Total	202	1,691 (31%)	5,322

takes a few seconds to less than 3 minutes, which is relatively trivial. We do not consider the execution time for dynamic analysis since *JaCoCo* is third-party software. However, based on *JaCoCo* developers, the performance overhead is approximately a 10% increase from normal test execution time (Jacoco, 2003). However, our static analysis help us locate inheritance-related issues and saves time for our dynamic analysis.

7.5 Results

In this section, we first introduce the studied systems. Then, we study inheritance-induced test redundancy by answering four RQs.

Studied Systems. We conduct our analysis on 15 open-sourced systems. The selection of 15 systems was influenced by time constraints, as analyzing all 503 systems from Section 7.2 would have been time-consuming. To identify the 15 systems, we applied additional criteria to identify highly maintained systems: the presence of inheritance, usage of *JUnit* in the *Maven* configuration files, sufficient test files, containing commit activity between 2022-2023, high popularity (stargazer count>600), and non-forked repositories. The criteria for systems to have inheritance is to ensure that redundancies are common issues for

systems that involve inheritance. From the pool of systems, we randomly selected the following 15 systems: Commons-math, Commons-lang, Iotdb, Maven, Pdfbox, Shiro, Shenyu, Biojava, Rocketmq, dubbo, Avro, Zookeeper, Commons-collections, Feign, and Graphhopper. As shown in Table 7.2, these studied systems cover different domains, from distributed databases to stream processing frameworks, message brokers, and group chat servers. Table 7.2 also displays the number of test inheritance extracted from the studied systems using the technique from Section 7.4.1. The results reveal that despite the relatively small number of test inheritances tree in some systems, like *commons-collections* (i.e., 10 inheritances), their impact on the codebase was substantial, constituting 85% of the total number of test classes. In contrast, in systems like *Avro*, which had a higher number of test inheritances tree (i.e., 46), the percentage of impacted test classes was lower, at 49%. This finding suggests that even a few instances of test inheritance can substantially influence overall testing structure. It underscores the need to analyze the intricacies inheritance in each system to grasp the true impacts of test inheritance. In conclusion, the results highlight the diverse nature of test inheritance in different systems.

7.5.1 RQ1: How Prevalent are Inheritance-Induced Redundant Test Candidates?

As discussed in Section 7.2, inheritance is widely adopted in practice, which raises an intriguing question about the number of test cases inherited from `superclasses` and whether test cases may become redundant. Particularly, when extensive test inheritance occurs, test cases may be inherited by numerous test `subclasses`, leading to challenges in understanding the test logic inherited from the `superclass` and the possibility of redundant test execution. This research question aims to investigate the occurrence of *Inheritance-Induced Redundant Test Candidates* to shed light on the implications of test inheritance in real-world projects. By exploring these test case relationships, we aim to gain valuable insights into the impact of inheritance in software testing.

Approach. In Section 7.4.1, we presented our static analysis approach to identify *Inheritance-Induced Redundant Test Candidates*. These candidates arise when two or more `subclasses`

inherit the same test cases from the `superclass`, leading to the execution of inherited test cases multiple times (may or may not be redundant). To assess the prevalence of these candidates, we compare them with all executable test cases in the studied systems, using the *mvn-surefire* test execution strategy by following the approach that is described in Section 7.4.3.

Result. ***On average, inheritance-induced redundant test candidates account for approximately 13% of the total executable test cases.*** This finding is based on a comparison of the number of *Inheritance-Induced Redundant Test Candidates* against the total number of executable test cases, which is reported in the last column, i.e., # Discovered Candidates, as shown in Table 7.3. Specifically, out of the 40,420 executable test cases in the examined systems, 5,080 (13%) are attributed to potential redundancies, specifically through test inheritance. This indicates a potentially significant impact of test inheritance on the overall testing efforts. For instance, Table 7.3 highlights that 50% of the test cases in *Commons-C.* are contributed through inheritance, followed by 21% in both *Commons-M.* and *Feign*. The results highlight the high prevalence of *Inheritance-Induced Redundant Test Candidates* among the test cases. While the average percentage of *Inheritance-Induced Redundant Test Candidates* may seem modest at 13%, the presence of such test cases is not negligible, given the large number of test cases in the examined systems. Furthermore, certain systems, such as *Commons-C.*, show a remarkably high percentage of inherited test cases, highlighting the importance of examining inheritance relationships and their impact on testing.

Initially, redundant test candidates come from 790 unique test cases. However, through inheritance the number of redundant candidates can increase sixfold. We also analyze the number of different `subclasses` from which the redundant candidates can be inherited. For instance, if a *Inheritance-Induced Redundant Test Candidates* test case is inherited from two `subclasses`, then we consider this test case to be multiplied two times through inheritance. We depict this multiplier in column 8 (e.g., Multiplier) in Table 7.3. Furthermore, we provide three summary statistics (mean, max, min) to show the diversity of inheritance. Hence, Table 7.3 shows that *Commons-C.* has 230 unique test cases

Table 7.3: Revealing the landscape of redundant test candidates:
A summary of the static analysis result.

Project	*Inheritance-Induced Redundant Test Candidates*						#Total Executable Test Cases
	#Unique Test Cases Defined in `Superclass`	#Occurrence in various subclasses			Multiplier	#Discovered Candidates	
		Mean	Max	Min			
commons-collections	230	14	93	2	14x	3167 (50%)	6367
commons-math	251	3	17	2	3x	866 (21%)	4036
feign	69	4	7	2	3x	299 (21%)	1441
shiro	14	5	11	2	5x	65 (8%)	822
dubbo	66	4	6	2	4x	236 (7%)	3626
biojava	30	3	3	2	3x	86 (6%)	1436
avro	31	4	5	2	4x	122 (3%)	3984
graphhopper	44	2	3	2	2x	98 (3%)	3347
maven	18	2	2	2	2x	36 (3%)	1058
shenyu	1	26	26	26	26x	26 (3%)	874
zookeeper	23	2	2	2	2x	46 (1%)	3156
pdfbox	4	3	4	2	3x	12 (1%)	1905
iotdb	5	2	2	2	2x	10 (1%)	1618
commons-lang	3	3	3	3	3x	9 (<1%)	6137
rocketmq	1	2	2	2	2x	2 (<1%)	613
Total	790	N/A	N/A	N/A	6x	5080 (13%)	40420

that are defined in `superclasses`. These test cases undergo inheritance through an average of 14 `subclasses`, with a maximum of 93 `subclasses` and a minimum of 2 `subclasses`. Through various inheritance practices, the number of redundant candidates then multiplies by 14x, increasing to 3167 test cases. Notably, all systems initially have a smaller subset of test cases (790). However, through inheritance, the total number of potential redundant candidates can increase sixfold (to 5,080). These initial findings underscore the potential of identifying and addressing redundant test candidates to optimize testing resources.

> **Answers to RQ1.** We discovered that 13% of the total executable test cases are *Inheritance-Induced Redundant Test Candidates*. These instances of redundancies are primarily caused by a small subset of test cases, but their occurrence increases six-fold through the inheritance process.

7.5.2 RQ2: Are the Inheritance-Induced Redundant Test Candidates Truly Redundant?

In RQ1, based on our static analysis results, we observed that systems that utilize inheritance consistently exhibit potential redundant candidates. However, assessing the redundancy of a test case solely through static analysis presents challenges. For example, consider a scenario where two subclasses inherit the same test case. Due to program dependencies, each subclass may have different execution contexts (e.g., through test fixtures (D. J. Kim, Tsantalis, et al., 2021)) of the test case differently; thus, these test cases cannot be considered redundant. Hence, in this RQ, our objective is to delve deeper into the true redundancy of these uncovered redundant candidates by conducting a comparative analysis of their code coverage and test oracles. By examining these quality attributes, we aim to gain comprehensive insights into the true redundancy of these test cases. We believe that this investigation will contribute significantly to enhancing the understanding of the effectiveness of these redundant test candidates and their overall impact on testing quality.

Approach. We define test case redundancy as the condition where the coverage and test

oracles are identical. We collect coverage and oracle following the approach from Section 7.4.3. For complete coverage comparison, we compare branch, line and instruction. We use Algorithm 2, to identify truly redundant tests from the initial ***Inheritance-Induced Redundant Test Candidates***. Note that, employing dynamic analysis to compare all test executions can be resource-intensive, as not all redundant test cases are induced by inheritance, which makes our static analysis an important intermediate step to focus on *Inheritance-Induced Redundant Test Executions*. The algorithm proceeds through three key steps. In step ①, a set of redundant test candidates, along with their corresponding coverage and oracle information, is provided as input. In step ②, as shown in *line 4*, the algorithm generates all possible pairwise combinations of the redundant candidates. For each pair, it compares both their coverage and oracle information. A pair is deemed redundant only when both coverage and oracle are found to be equivalent. This comparison is represented by a triplet, denoted as $< Test1, Test2, Boolean >$, where the boolean value is `True` if and only if both coverage and oracle are equal, and `False` otherwise. Given that pairwise comparisons are employed among the redundant candidates, the number of comparisons performed follows the formula $\frac{n!(n-r)!}{r!}$. In step ③, as shown in *line 5*, we use *union-find* (Wikipedia, 2023b) algorithm to establish the connected component relationships within the pairwise comparisons. The output of the relationship is denoted as *Group*. In step ④, as shown in *line 7-12*, we examine the *Group* and flag component that has at least two candidates to be redundant, i.e., have the same code coverage and oracle. For example, given $< A, B, False >$, $< A, C, False >$, and $< B, C, True >$, the algorithm will flag that the presence of A always leads to *False*, indicating that B and C are redundant, while A is non-redundant.

Result. ***45% of identified redundant test candidates are truly redundant.*** Table 7.4 provides an overview of the identified redundant test executions among the studied redundant test candidates. We uncover that the majority of systems exhibit redundant tests, with around 50% or more of the candidates falling into redundant tests in most systems. Notably, *Commons-C.* initially contained the highest number of redundant test candidates. However, through redundancy analysis, a significant portion of these candidates (86%) were

Algorithm 2 Redundancy Analysis

```
Input: Array Coverage, Array Oracle
Output:
 1: Global Var1: RedundantTest
 2: Global Var2: noRedundantTest
 3: procedure findRedundantTest(Coverage,Oracle)
 4:     Pairs ← doPairWiseCombination(Coverage, Oracle)
 5:     Group ← unionFind(Pair)
 6:     for group in Groups do
 7:         if len(group) > 1 then
 8:             RedundantTest ← group
 9:         else
10:             noRedundantTest ← group
11:         end if
12:     end for
13: end procedure
```

identified as non-redundant, leaving only 14% as truly redundant. This may be related to developers' design and usage of inheritance in test cases. For example, in *Commons-C.* developers commonly use test inheritance to help test diverse implementations of different algorithms, e.g., sorting algorithms. Namely, these sorting algorithms share the same test setup, i.e., create new lists, prior to testing the sorting algorithm. Developers use test inheritance to reuse code and avoid duplication.

We uncover that on average 45% of the redundant test candidates are truly redundant when considering both code coverage and test oracles, while the remaining 55% demonstrate differences in either code coverage, test oracles, or both, making them non-redundant. These results yield two important insights: Firstly, the significant presence of redundancy (45%) among the identified candidates of redundant test cases suggests opportunities for eliminating tests that may not contribute to fault localization. Secondly, from another perspective, the presence of a significant number (55%) of non-redundant test cases highlights the potential benefits of test case inheritance in not only facilitating code reuse but also diversifying code coverage and assertions, enhancing testing practice.

Answers to RQ2. We uncover that 45% of the redundant test candidates are truly redundant tests, whereas the remaining test cases facilitate diversification of coverage and assertion oracle.

7.5.3 RQ3: How Much Time does Inheritance Induced Redundant Test Contribute to the Overall Test Execution Time?

As seen in RQ2, there is a large occurrence of redundant test cases exhibiting identical coverage and oracle results. Considering the prevalence of these redundant tests, it is important to investigate their impact on the overall test execution time and to what extent to which these redundant test cases prolong the testing process.

Approach. Following Section 7.4.3, we employed *Maven-Surefire* to execute the redundant test cases and collect the test execution time. Specifically, we use `mvn clean test` to run all the test cases and `mvn clean test -DTest=RedundantCandidates` to sequentially run redundant test cases for each studied project within a single JVM. Note that the test execution time excludes the time for code compilation. To enhance the reliability of our findings, we conducted data collection five times and calculated the average results.

Result. ***On average, the detected inheritance induced redundant tests contributes to 14% of the total test execution time.*** Table 7.4 provides detailed insights into the execution time of redundant tests and the total number of tests for different studied systems. Among the studied systems, 11 out of 15 systems contained redundant test cases. The presence of these redundant tests had a large impact on the overall test execution time, constituting approximately 14% of the total time. As anticipated, the extent to which redundant test cases contributed to the execution time was closely related to their proportion within the total number of test cases. For example, systems such as *Feign* and *Commons-C.* exhibited a higher overall execution time due to a substantial number of redundant test cases. Interestingly, for *Commons-C.*, although it has many redundant candidates (e.g., 2677), it has fewer truly redundant tests (e.g., 385), which is the opposite for other systems like *Feign*, where 100% of candidates are truly redundant. More importantly, we find that

Table 7.4: The prevalence of redundant tests and their impact on test execution time. *Candidates* refers to *Inheritance-Induced Redundant Test Candidates* and *Redundant* refers to true redundant test cases.

Project	**# Test Execution**		**Test Execution Time (seconds)**	
	Candidates	**Redundant**	**Total Tests**	**Redundant**
Feign	291	291 (100%)	183.996	146.428 (79.6%)
Commons-C.	2677	385 (14%)	16.076	4.373 (27.2%)
Avro	106	87 (82%)	215.396	36.988 (17.2%)
Shiro	65	30 (46%)	86.845	13.696 (15.8%)
Commons-M.	817	415 (51%)	75.371	5.377 (7.1%)
Rocketmq	2	2 (100%)	267.950	1.974 (0.7%)
Pdfbox	12	8 (67%)	66.070	0.257 (0.4%)
Biojava	86	63 (73%)	774.207	2.615 (0.3%)
Iotdb	10	4 (40%)	1362.134	3.540 (0.3%)
Maven	36	18 (50%)	48.927	0.103 (0.2%)
Dubbo	201	99 (49%)	1322.386	0.493 (0.1%)
Commons-L.	9	0	N/A	N/A
Graphhopper	92	0	N/A	N/A
Shenyu	26	0	N/A	N/A
Zookeeper	42	0	N/A	N/A
Average	% Redundancy in Candidates 45%		% Contribution to Execution Time 14%	

while *Commons-C.* has a lower percentage of truly redundant tests (e.g., 17.2%) compared to *Commons-M.* and *Avro*, its redundant test execution contributes much more to the overall execution time. Therefore, the impact of redundant tests depends on the test design, and some systems may be more affected by test execution time. This finding also underscores the potential benefits of removing redundancy, as it has the potential to significantly improve testing resources.

Answers to RQ3. We uncover that 14% of total test execution time is spent on redundant test cases that do not provide any additional benefits.

Discussion. Expanding upon the findings of our RQ3, it becomes evident that using inheritance in the test code leads to an increased occurrence of redundant test execution. Such redundancies are not immediately noticeable in the test code, as they can be inherited from its superclass. Our methodology allows for the detection of such redundancies caused by

inheritance, providing developers with awareness of potential bottlenecks in test code. Developers can bypass these redundancies by using the build system (Maven) to exclude test cases from execution. Another bypassing strategy is to override the inherited test cases with another test case in the subclass annotated with @Ignore (Apache Ignite - 63b9e1653d), as often shown in prior work (D. J. Kim, Yang, et al., 2021). However, these strategies do not completely remove redundancies in the test code; they only skip their execution, overlooking the complexity of redundancy removal. In RQ4, we elaborate on the complexity of removing redundancies related to inheritance test cases.

7.5.4 RQ4: Assessing the Feasibility of Reducing Inheritance-Induced Redundant Test Execution?

In prior RQs, we uncovered many test redundancies. Consequently, the next natural step is to eliminate these redundancies that do not contribute effectively to fault localization capabilities in order to improve test execution time. While developers could temporarily bypass such tests, the removal of redundant test cases within an inheritance context presents a more significant challenge. As observed in RQ1, in extreme cases, a test case can be inherited and executed as many as 96 times, demonstrating complex coupling and making the task of redundancy removal challenging. Hence, in this RQ, we conduct an empirical analysis to understand the feasibility of removing redundant test cases in inheritance. Our aim is to provide insights to aid future research in the development of test case minimization tools.

Approach. We conduct a feasibility analysis because, unlike previous works on test case minimization that typically involve straightforward removal of redundant test cases (Di Nardo et al., 2015; Leitner, Oriol, Zeller, Ciupa, & Meyer, 2007; Pan, Ghaleb, & Briand, 2023), the scenario of test case redundancies related to inheritance contains complex coupling and necessitates careful refactoring decisions. Hence, we conduct two analysis to study the challenges of removing inheritance induced redundant test cases. We list them below:

- *RQ4(A). Can inherited test cases become both redundant and non-redundant test*

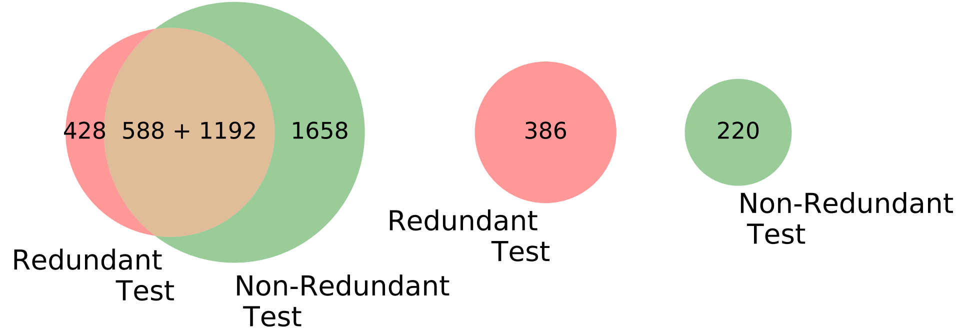

Figure 7.3: Overlap between Redundant and Non-redundant Tests in Test Execution. The overlapping region (orange) indicates that an inherited test case is redundant in one `subclass` but non-redundant in another `subclass`. The one figures on the left correspond to five systems, while the two figures on the right correspond to the rest.

executions?

- *RQ4(B). How far are the redundant test cases from their definition of the superclasses in the inheritance trees?*

RQ4(A): Can inherited test cases contain both redundant and non-redundant groups of tests?

In RQ3, We uncovered that 45% of identified redundant candidates are truly redundant, whereas the remaining is non-redundant, i.e., through code coverage or test oracle. In this RQ, we hypothesize that it is possible for inherited test cases to be redundant in one context, i.e., redundant in one `subclass` but not redundant in another `subclass`. The existence of such a complex scenario will give us an initial glimpse of the challenges for efficient test case minimization.

Approach. We modify Algorithm 2 to check if the equivalent group contains both redundant test cases and non-redundant test cases resulting from the same *Inheritance-Induced Redundant Test Candidates*, and denote this as *Co-existence* group.

Result. ***Out of 1,402 detected redundant tests, 588 (41%) test cases co-occurs with non-redundant tests.*** As illustrated in Figure 7.3, inherited tests can result in both redundant and non-redundant test executions. In other words, inheriting a test case

results in redundancies in one `subclass`, but not in another `subclass`, due to the different execution contexts specified by developers, e.g., through test fixtures. Specifically, our analysis reveals that among the 1,402 detected truly redundant test cases, 588 test cases actually co-exist with their non-redundant test case counterparts. Notably, 5 out of 15 studied systems (i.e., *Avro*, *Commons-C.*, *Commons-M.*, *Dubbo*, and *Shiro*) encompass this co-existence of redundancy and non-redundancy in the inherited test cases. This suggests that even for the same test case defined in a superclass, inheritance of this test case does always cause redundancy, and some may be utilized in different `subclass` contexts (e.g., to ease maintenance and improve coverage). However, whether test inheritance is beneficial to test design remains a future research problem. While it may improve code coverage, it can also increase code complexity, which may become difficult to maintain in the long run. Nonetheless, the variability in the nature of redundancy may be related to the design of test cases in different systems. These findings highlight a complex scenario for effective test case minimization.

RQ4(B): How far are the redundant test cases from their definition of the superclasses in the inheritance trees?

As seen in RQ4(A), *Inheritance-Induced Redundant Test Candidates* can be found in many different `subclass` contexts, co-existing with non-redundant test cases. In this RQ, we delve into the distance of these executable test cases in the `subclass`, i.e., where the test case is executed, from their `superclass`, i.e., where the test case is declared. Analyzing class distance will reveal the potential existence of complex hierarchical relationships, i.e., how many `subclasses` do these inherited test cases impact through inheritance? Analyzing such distance is beneficial to understand the challenges of removing redundant tests that impact multiple classes.

Approach. We investigate class distance in all of the *Redundancy-Inducing Inheritance* (e.g., 4,472 test candidates), including redundant test, non-redundant test, and co-existence of both. To analyze class distance, we leverage our inheritance tree from Section 7.4.1. We use the *shortest-path* algorithm (Wikipedia, 2023a) to find the path it takes to reach the

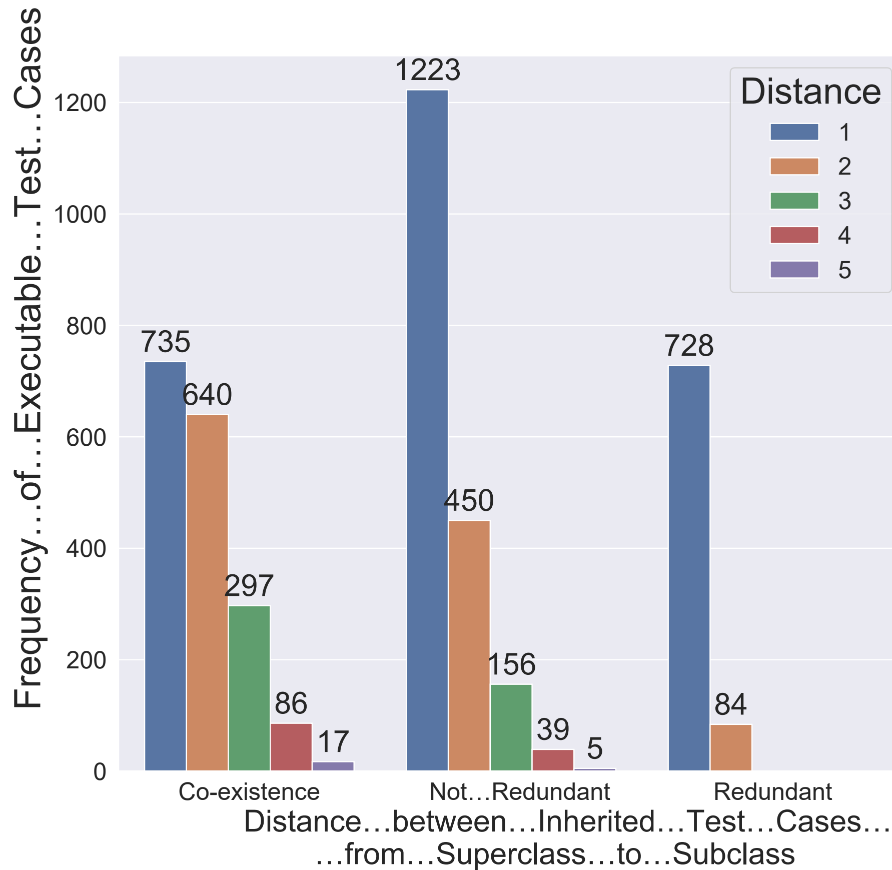

Figure 7.4: Analysis of distance in the inheritance tree between the parent test case and the child test.

impacted `subclass` from a `superclass`.

Result. ***Inherited test cases that lead to both redundant and non-redundant test executions in `subclasses` exhibit a highly variable number of inheritance distance from `superclass`.*** As shown in Figure 7.4, majority of redundant tests (728/812 - 90%) are executed by the direct `subclass`, whereas remaining 10% have executions that executed by two `subclasses` downstream. The finding shows that these redundant test cases may be easier to resolve. Interestingly, we find that for tests that co-exist with non-redundant tests, there are more diverse sets of class distances. In particular, some non-redundant tests may be executed up to five class distances in the downstream `subclass`. Namely, for systems *Avro*, *Commons-C.*, *Commons-M.*, *Dubbo*, and *Shiro*, which contains co-existence of redundant and non-redundant tests, there is a more complex inheritance distance. This reveals that inheritance relationships within *Inheritance-Induced Redundant Test Executions*

may have significant variability in hierarchical structures. Specifically, it is possible that test cases designed with such complexity in class distance are less likely to be redundant, as they impact a higher number of `subclasses`, whereas much simpler class hierarchies have a higher tendency to be redundant. Nonetheless, the presence of a complex hierarchy constitutes additional complexity that makes test case minimization challenging, as it may impact many downstream `subclasses`.

Answers to RQ4. Removing redundant tests need careful preservation of code coverage. This is particularly important when dealing complex inheritance relationships, where co-existence of both redundant and non-redundant can contribute to code coverage and may impact multiple classes.

Discussion. Expanding upon the findings of our study, which demonstrate a significant overlap between redundant and non-redundant test cases, it becomes evident that the removal of redundant test cases, as often seen in traditional test case reduction strategies, may not represent a valid strategy for inherited test cases. Our results underscore the complexity of the issue. In other words, while redundancies must be addressed, it is also apparent that certain test cases leverage inheritance to enhance coverage and assertions, indicating their value in the testing process. Consequently, this raises an interesting question: How can we reorganize the test interfaces to reduce redundancies and improve test maintainability? Namely, our results show the possibility of exploring higher-level architectural refactoring to enhance test quality. Nevertheless, our approach provides initial insights to eliminate the impact on test execution overhead.

7.6 Threats to Validity

Internal Validity. Firstly, our findings depend on the accuracy of the third-party tool (e.g., *spoon*) to mine *Redundancy-Inducing Inheritance* and *Inheritance-Induced Redundant Test Candidates* in the source code and also the accuracy of the dynamic analysis tool (e.g., *Jacoco*) to execute *Inheritance-Induced Redundant Test Executions*. It is important

to note that validating the precision of these third-party tools is not within the scope of our responsibility, However, both *spoon* and *Jacoco* are widely used in prior research and in practice and we did not find any false positives during our manual examination of the results.

External Validity. Our studied systems are all open source systems implemented in Java, so the result may not be generalized to all systems. However, to minimize the threat, we follow a set of criteria to popular systems from various domains, large in scale, and actively maintained. Within this criteria we randomly sample 15 studied systems to obtain diverse studied systems. However, we acknowledge that the three projects containing over 85% of the redundancy candidates might indicate a concentration of the issue in certain projects. Nevertheless, the representatives of the entire open-source Java project ecosystem is a complex matter. Our intent was not to claim that this issue is uniformly distributed across all projects but to highlight that our findings are dependent on how different systems use inheritance in their test code design. Hence, our work may not be applicable to all systems, and the impact may be more significant in larger projects. Although our tool is designed for analyzing Java systems, we have made our source code available, where our implementations may inspire writing similar analyses for other programming languages. We encourage future studies to replicate our experiments on other systems and projects implemented in different programming languages.

Construct Validity. Our dynamic analysis encountered some test failures and environmental errors, resulting in un-executed test cases and potentially under-representing our analysis for *Inheritance-Induced Redundant Test Executions*. However, the number of un-executed tests is small. There may be bugs in the tools that we use. For example, prior to *JaCoCo* version 0.8.10 (i.e., most updated version), the *report-aggregate* plugin contained a bug where it only collects coverage of dependent module except for its current module (Jacoco, 2013). We noticed the issue and migrated to the fixed version of Jacoco. However, there may still be undiscovered bugs in the tools that can affect the results. Our technique leverage functionality from third-party software, such as *Spoon* and *Jacoco.* We leverage *Spoon* to extract *Redundancy-Inducing Inheritance* and *Inheritance-Induced Redundant*

Test Candidates, whereas we leverage *Jacoco* to identify *Inheritance-Induced Redundant Test Executions*. Moreover, for extracting assertion of the test cases we also rely on the *Spoon* API. It is important to note that validating the precision of these third-party tools is not within the scope of our work. However, our manual investigation of the results from *Redundancy-Inducing Inheritance* achieved 100% precision.

7.7 Implications & Contribution

Based on our empirical findings, we present actionable implications and future research directions for researchers and practitioners.

7.7.1 Discussion and Implication for Researchers.

R1: Future research should explore test removal while preserving code coverage, as inheritance-induced redundant test cases may overlap with non-redundant tests. While our analysis revealed many *Inheritance-Induced Redundant Test Executions*, in RQ4 we also found a 41% overlap with tests that contribute to increased/different code coverage. This presents a challenge in determining how to remove redundant test executions while preserving non-redundant test cases that aim to increase code coverage. The co-existence of redundant and non-redundant tests complicates test case reduction, as both types of tests serve different purposes. Redundant test cases may increase execution time and hinder fault localization capability, while non-redundant test cases play a role in increasing code coverage. Hence, future research is necessary to comprehend the trade-offs associated with using inheritance to achieve code coverage and its potential increase in redundancy.

R2: Further research may investigate trade-offs between using inheritance to make tests reduce maintenance cost and not using inheritance to reduce test case redundancies. While inheritance in test (apache dev, 2023; stackoverflow, 2013a, 2020) is a controversial practice, we find that 40% amongst 503 sampled systems utilize inheritance in test code, which is widely adopted in practice. In particular, projects like Commons-collections and Commons-math, despite their heavy reliance on inheritance, exhibit fewer

redundancies, hinting at the compactness and superior quality of their tests. This opens the door to future research avenues, exploring the trade-offs between employing inheritance and abstaining from it. Future research may also delve into quality attributes, such as the time required for activities like bug fixing, coverage enhancement, and feature addition, comparing tests that employ inheritance to those that do not. Moreover, for test cases that result in redundancies, future studies may also investigate how they manifest in the code and provide preventative measures.

R3: In general, future research is needed to understand how to remove complex inheritance relationships in the test code. In RQ4, we revealed that redundant tests may exhibit complex inheritance hierarchy relationships. Removing redundant tests in such scenarios poses a challenge, as redundancy impacts multiple class relationships. Further research is needed to explore effective strategies and tools to refactor these complex inheritance relationships in general, which may also help remove inheritance-induced redundant test cases. Namely, our paper show the possibility of exploring higher-level architectural refactoring to enhance test quality.

R4: We uncovered the widespread existence of inheritance-induced redundant test cases. How these test cases impact fault localization can be further explored in future research. As redundant test cases are inherited from other test classes, test failures may be difficult to localize using fault localization. For instance, in *Apache-Avro*, the *TestProtocolSpecific* class contains 15 test cases that are inherited and executed by five different *subclasses*. Interestingly, all 15 test cases fail in one *subclass* while passing in the remaining four, which might be due to specific bugs associated with the test setup in that particular *subclass*. As these failures are not indicative of source code defects, they could potentially mislead developers and fault localization algorithms, which attempt to localize source code defects (Wong, Gao, Li, Abreu, & Wotawa, 2016), causing them to identify faults incorrectly. We encountered a similar scenario in *AbstractOrderedBidiMapDecoratorTest* from *Commons-collections*. Considering that *Commons-collections* is part of the *defects4j* benchmark and contains many inheritance-induced dependencies, future studies could also investigate how eliminating such redundancies can improve fault localization techniques

focus on distinct failure.

7.7.2 Discussion and Implication for Practitioners.

P1: Practitioners need better support for detecting repetitive test candidates. Inheritance is a double-edged sword, while it may improve test compactness and maintainability, it can also introduce test case redundancies. For example, as seen in RQ2, while many redundant test cases are caused by inheritance, they are related to a small subset of parent test cases. Furthermore, some test cases may repeat up to 93 times due to inheritance. Therefore, it would be beneficial to raise awareness among developers about these issues. Future work should provide tools to assists developers to be aware of the redundant test cases.

7.8 Chapter Summary

This paper presents the first empirical study on test case redundancy caused by inheritance. We propose a hybrid approach that combines static and dynamic analysis to detect and verify inheritance-induced redundant test cases. We apply our approach to 15 open-source Java systems. We find that (1) Despite controversies surrounding test inheritance, non-negligible tests (14%) of test case executions are redundant. (2) The redundant test cases take, on average, 13% of the total execution, which adds additional test execution overhead. (3) Many inherited test cases (40%) are redundant in some *subclass* but non-redundant in others, making it difficult to eliminate redundancy while preserving code coverage. This complexity calls for careful refactoring decisions to address the issue effectively. Finally, we also discuss challenges and future research directions on resolving inheritance-related issues.

Chapter 8

Conclusion and Future Work

This chapter summarizes the main ideas that are presented in this thesis. In addition, we propose future work to leverage program analysis to help improve test quality and maintainability.

8.1 Thesis Contribution

Despite extensive research on various aspects of test automation, including automated test prioritization, fault localization, and program repair, the design of tests remains an under-explored area. The software industry needs robust standards for test design and maintenance, which can significantly enhance overall software quality. This thesis aims to improve the quality of test code through effective design and maintenance practices in three aspects: 1) Re-evaluate current knowledge of test design issues and analyze their applicability in software industries, 2) Provide insights on test design practices based on modern software test automation frameworks to assist developers who may be facing similar issues in their software projects, and 3) Understand reusability in test code design and its impact on the effectiveness of test code.

For (1), we have one research outcome aimed at improving the current perception and

applicability of *"test smell"* (D. J. Kim, Chen, & Yang, 2021b). This study includes a manual classification of real-world open-source projects, providing a comprehensive catalogue of why and how developers resolve test smells during software evolution. We also investigate whether test smell addition or removal practices impact defect density. Our findings indicate that developers actively remove only certain types of test smells, and many test smells may not have a significant relationship with post-release defects, as shown by our explainable logistic regression model. Our study provides a systematic understanding of practitioners' perceptions of *"test smell"*, which could be valuable for developing automated tool supports.

For (2), we have two research outcomes aimed at understanding how the usability and re-configurability of modern automated testing frameworks impact test design. Our first empirical study focuses on the JUnit testing framework (D. J. Kim, Tsantalis, et al., 2021). We examine how annotation metadata can improve test design and identify annotation misuses that negatively impact test design. Additionally, we provide a comprehensive classification of test design practices that will serve as a valuable insights for developers. In the second empirical study, we delve into a controversial annotation API, @Ignore, which enables developers to disable tests to facilitate continuous deployment (D. J. Kim, Yang, et al., 2021). We propose an annotation miner capable of tracking the lifecycle of tests that undergo multiple rounds of disabling and re-enabling, not only through @Ignore annotations but also through code comments. Our objective is to understand why tests are disabled and re-enabled, aiming to provide better insights on maintainability challenges that may arise during testing process. Our classification on reasons behind why test become disabled and re-enabled reveals that while hard-to-fix bugs are the majority of the culprits, many tests remain disabled without validating software quality. Moreover, ad-hoc usage of disabling suggests that maintaining software quality involves trade-offs: if a bug can be fixed quickly, the test should be re-enabled promptly. However, we find that due to poor issue tracking practices for disabled tests, many tests remain disabled.

For (3), we present one research outcome aimed at detecting how using inheritance to achieve test case reusability can lead to redundancies in test execution. Our study demonstrates that inheritance is commonly used to achieve reusability in test code. We

provide an approach to assess the prevalence of redundancies caused by test inheritance.

In conclusion, we summarize the contributions of this thesis as follows:

- We proposed a classification for why developers remove test smells and their effects on software quality, particularly in terms of defect-proneness. We also build regression model to find the relationship between test smell addition/removal and software quality, in terms of defect-proneness (Chapter 4).
- We performed an empirical study on how developers leverage test annotations to maintain test code quality (e.g., readability, test flakiness, test performance, obsolete test). We proposed classification on usage and misusages of test annotations (Chapter 5).
- We performed an empirical research on the prevalence, evolution, and maintenance of disabling tests in practice. Studying why developers disable test code may help developers understand the source of potential technical debt in the test code that can direct future test maintainability practice (Chapter 5).
- We propose a hybrid approach based on static and dynamic test analysis to identify and locate test case redundancies introduced by inheritance. These redundancies can increase test execution time without additional fault-revealing capabilities. Our work suggests future research directions to provide better guidance to developers on using inheritance to prevent pitfalls such as redundancies (Chapter 6).

8.2 Future Work

This thesis represents a significant step towards improving test design and maintainability practices. However, there are numerous open challenges and research opportunities that can complement this work and contribute to developing comprehensive guidelines for test maintainability. Our main contribution is that, by enhancing test quality, these efforts can ultimately improve software quality as a whole. Hence, we discuss some potential directions for future work.

Providing complete automatic test design issue detection, recommendation and refactoring suggestions in Java-based system. Previous studies aimed to identify issues within test design and maintenance using a decade-old research perspective on *"test smell"*. In this thesis, we not only re-evaluate *"test smell"* (Deursen et al., 2001), but also propose several research directions for assessing the impact of modern test automation tools on test design practices. In this thesis, we have uncovered test design practices adopted by developers in open-source projects to provide insights on how to improve test design and maintenance practices. However, there remains a lack of automated tool support integrated into IDE solutions. Future research will explore recommendations and automated refactoring tools to better assist developers in maintaining test quality.

Extending test design issues in large-scale proprietary, cross-domain, and cross-language systems. In this thesis, we take a preliminary step in investigating test design issues stemming from the JUnit in open-source projects. We find that while JUnit can enhance test maintainability, there are instances of ad-hoc usage and misuse due to the configurable nature of its annotation APIs. Looking forward, our aim is to explore whether proprietary software systems face similar or different challenges, and whether they could benefit from recommendation or refactoring tools. Moreover, we aim to perform user studies to evaluate whether our developers could truly benefit from automated tool supports that aims to improve test maintenance from Junit framework.

Provide new benchmark on test code generation adhering best practices. Previous studies on automatic test suite generation using Evosuite (Fraser & Arcuri, 2011) and Randoop (Pacheco & Ernst, 2007) have demonstrated that these tools can achieve high test quality by attaining both high coverage and mutation scores, as well as the ability to detect faults (Almasi et al., 2017; Fraser & Arcuri, 2015). Despite this, generated unit tests pose a significant maintenance burden, making them harder to debug when included in a project (Ceccato et al., 2015), due to their poorer readability and maintainability compared to human-written counterparts.

Moreover, with the rise of large language models and the increasing ease of code generation tasks, it becomes paramount to validate the quality of generated code, specifically

test code in the context of the thesis. In future studies, we may provide benchmarks for techniques that validate the quality of automatically generated test code, assessing factors such as extensibility, maintainability, and readability.

Provide empirical study on the trade-offs between re-usability and maintainability. As seen in Chapter 6, while inheritance enables test re-usability and extensibility, it also leads to redundancy. However, some projects, despite heavily relying on inheritance, exhibit fewer redundancies, suggesting the compactness and superior quality of their test code. Future research should investigate how inheritance manifests in test code and determine when it should be used to improve maintainability or avoided to prevent design issues. Specifically, since inheritance can be a source of test debt by achieving re-usability at the expense of redundancy, future studies should also explore the best ways to achieve re-usability in test code.

Appendix A

Supplementary Figure

Table A.1: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot. χ^2	AUC (%)
Accumulo	BASE	LOC	33.05%	188.85	0.68
		pre_bug	20.8%		
		deletedLine	21.54%		
		fileChurn	8.52%		
		ts_coupling	13.57%		
		tt_coupling	2.52%		
	BASE+PD	ts_coupling	40.5%	63.25	0.71 (3.9%)
		tt_coupling	7.52%		
		Conditional.Test.Logic	6.11%		
		Constructor.Initialization	16.76%		
		Redundant.Assertion	6.75%		
		Eager.Test	6.17%		
		Duplicate.Assert	7.07%		
	BASE+PD+PR	ts_coupling	31.71%	80.79	0.73 (6.4%)
		tt_coupling	5.89%		
		Conditional.Test.Logic	4.78%		
		Constructor.Initialization	13.12%		
		Redundant.Assertion	5.28%		
		Eager.Test	4.83%		
		Duplicate.Assert	5.54%		
		Duplicate.Assert.Added	4.95%		
		Unknown.Test.Added	10.78%		
		Eager.Test.Removed	5.98%		
Bookkeeper	BASE	LOC	56.96%	177.81	0.8
		pre_bug	5.12%		
		codeChurn	22.63%		
		fileChurn	4.8%		
		ts_coupling	10.48%		
	BASE+PD	ts_coupling	25.04%	74.40	0.84 (5.3%)
		Assertion.Roulette	9.16%		
		Constructor.Initialization	20.52%		
		Lazy.Test	5.54%		
		Unknown.Test	23.86%		
		Resource.Optimism	15.89%		
	BASE+PD+PR	ts_coupling	16.61%	112.12	0.87 (8.8%)
		Assertion.Roulette	6.08%		
		Constructor.Initialization	13.62%		
		Lazy.Test	3.67%		
		Unknown.Test	15.83%		
		Resource.Optimism	10.54%		
		Exception.Catching.Throwing.Added	4.4%		
		Lazy.Test.Added	3.75%		
		Unknown.Test.Added	7.85%		
		Magic.Number.Test.Added	5.82%		
		General.Fixture.Removed	3.85%		
		Sleepy.Test.Removed	7.97%		

Table A.2: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot. χ^2	AUC (%)
Camel	BASE	LOC	76.33%	276.76	0.74
		pre_bug	4.76%		
		codeChurn	7.79%		
		ts_coupling	11.12%		
	BASE+PD	ts_coupling	34.6%	88.98	0.75 (1.3%)
		Assertion.Roulette	15.11%		
		Conditional.Test.Logic	6.02%		
		General.Fixture	8.2%		
		Mystery.Guest	14.6%		
		Redundant.Assertion	8.01%		
		Duplicate.Assert	7.82%		
		Resource.Optimism	5.64%		
	BASE+PD+PR	ts_coupling	31.38%	104.29	0.75 (1.6%)
		Assertion.Roulette	12.57%		
		Conditional.Test.Logic	5.59%		
		General.Fixture	7.23%		
		Mystery.Guest	12.42%		
		Redundant.Assertion	5.82%		
		Duplicate.Assert	5.81%		
		Resource.Optimism	4.86%		
		Print.Statement.Added	4.47%		
		Print.Statement.Removed	4.86%		
		Eager.Test.Removed	4.99%		
Cassandra	BASE	LOC	43.38%	228.50	0.75
		pre_bug	33.02%		
		codeChurn	6.37%		
		fileChurn	8.95%		
		ts_coupling	5.72%		
		tt_coupling	2.57%		
	BASE+PD	ts_coupling	69.01%	18.93	0.76 (1.1%)
		tt_coupling	30.99%		
	BASE+PD+PR	ts_coupling	30.73%	42.50	0.78 (3.3%)
		tt_coupling	13.8%		
		IgnoredTest.Added	11.15%		
		Conditional.Test.Logic.Removed	33.06%		
		Exception.Catching.Throwing.Removed	11.25%		

Table A.3: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot. χ^2	AUC (%)
Flink	BASE	LOC	46.04%	1396.14	0.76
		pre_bug	35.08%		
		codeChurn	6.61%		
		deletedLine	0.29%		
		fileChurn	10.41%		
		ts_coupling	1.56%		
	BASE+PD	ts_coupling	19.61%	111.28	0.77 (1.0%)
		Conditional.Test.Logic	9.81%		
		Exception.Catching.Throwing	4.96%		
		General.Fixture	12.99%		
		Mystery.Guest	22.05%		
		Lazy.Test	3.62%		
		Unknown.Test	20.69%		
		Magic.Number.Test	6.27%		
	BASE+PD+PR	ts_coupling	16.27%	134.14	0.77 (1.5%)
		Conditional.Test.Logic	8.14%		
		Exception.Catching.Throwing	4.12%		
		General.Fixture	10.77%		
		Mystery.Guest	18.29%		
		Lazy.Test	3.0%		
		Unknown.Test	17.16%		
		Magic.Number.Test	5.2%		
		EmptyTest.Added	4.28%		
		Mystery.Guest.Added	3.67%		
		Duplicate.Assert.Removed	3.08%		
		Magic.Number.Test.Removed	6.01%		
Zookeeper	BASE	LOC	50.8%	79.69	0.79
		pre_bug	36.45%		
		codeChurn	12.75%		
	BASE+PD	Resource.Optimism	100.0%	4.90	0.8 (1.7%)
	BASE+PD+PR	Duplicate.Assert.Removed	25.57%	16.76	0.83 (5.4%)
		Unknown.Test.Removed	74.43%		

Table A.4: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot χ^2	AUC (%)
Hive	BASE	LOC	72.11%	500.82	0.64
		pre_bug	1.76%		
		codeChurn	6.96%		
		ts_coupling	5.67%		
		tt_coupling	13.5%		
	BASE+PD	ts_coupling	14.86%	191.14	0.66 (1.6%)
		tt_coupling	35.38%		
		Conditional.Test.Logic	2.58%		
		EmptyTest	3.62%		
		General.Fixture	9.66%		
		Lazy.Test	12.53%		
		Duplicate.Assert	4.16%		
		Unknown.Test	9.21%		
		IgnoredTest	2.09%		
		Resource.Optimism	5.91%		
	BASE+PD+PR	ts_coupling	10.94%	259.52	0.67 (3.3%)
		tt_coupling	26.05%		
		Conditional.Test.Logic	1.9%		
		EmptyTest	2.67%		
		General.Fixture	7.12%		
		Lazy.Test	9.23%		
		Duplicate.Assert	3.07%		
		Unknown.Test	6.78%		
		IgnoredTest	1.54%		
		Resource.Optimism	4.35%		
		Conditional.Test.Logic.Added	1.56%		
		Mystery.Guest.Added	3.57%		
		Duplicate.Assert.Added	5.94%		
		IgnoredTest.Added	1.84%		
		Redundant.Assertion.Removed	2.27%		
		Sensitive.Equality.Removed	2.84%		
		Eager.Test.Removed	2.08%		
		Duplicate.Assert.Removed	1.62%		
		Unknown.Test.Removed	3.15%		
		Magic.Number.Test.Removed	1.48%		
Wicket	BASE	LOC	89.35%	103.02	0.78
		fileChurn	6.56%		
		tt_coupling	4.09%		
	BASE+PD	tt_coupling	14.08%	29.95	0.8 (2.9%)
		Conditional.Test.Logic	48.84%		
		Eager.Test	23.95%		
		Unknown.Test	13.13%		
	BASE+PD+PR	Conditional.Test.Logic	22.75%	48.82	0.82 (5.6%)
		Eager.Test	12.09%		
		Unknown.Test	7.92%		
		Assertion.Roulette.Added	30.48%		
		Conditional.Test.Logic.Added	17.51%		
		Exception.Catching.Throwing.Added	9.26%		

Table A.5: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot χ^2	AUC (%)
Kafka	BASE	LOC	58.84%	848.39	0.8
		pre_bug	28.83%		
		codeChurn	2.0%		
		fileChurn	4.69%		
		ts_coupling	4.52%		
		tt_coupling	1.13%		
	BASE+PD	ts_coupling	31.14%	123.05	0.81 (1.6%)
		tt_coupling	7.76%		
		Exception.Catching.Throwing	19.13%		
		Mystery.Guest	10.74%		
		Redundant.Assertion	4.45%		
		Lazy.Test	9.81%		
		Duplicate.Assert	6.8%		
		Unknown.Test	4.14%		
		Magic.Number.Test	6.02%		
	BASE+PD+PR	ts_coupling	28.13%	136.23	0.82 (2.6%)
		tt_coupling	7.01%		
		Exception.Catching.Throwing	17.28%		
		Mystery.Guest	9.7%		
		Redundant.Assertion	4.02%		
		Lazy.Test	8.86%		
		Duplicate.Assert	6.14%		
		Unknown.Test	3.74%		
		Magic.Number.Test	5.44%		
		Exception.Catching.Throwing.Added	3.57%		
		Exception.Catching.Throwing.Removed	2.82%		
		Print.Statement.Removed	3.28%		
Karaf	BASE	LOC	59.15%	10.26	0.64
		tt_coupling	40.85%		
	BASE+PD	tt_coupling	9.8%	42.78	0.75 (14.4%)
		General.Fixture	14.16%		
		Print.Statement	16.85%		
		Sleepy.Test	47.66%		
		Unknown.Test	11.53%		
	BASE+PD+PR	tt_coupling	6.16%	72.25	0.82 (28.4%)
		General.Fixture	8.53%		
		Print.Statement	9.76%		
		Sleepy.Test	30.24%		
		Unknown.Test	6.33%		
		Eager.Test.Added	5.87%		
		IgnoredTest.Added	9.52%		
		Magic.Number.Test.Added	9.87%		
		General.Fixture.Removed	13.73%		
Hadoop	BASE	LOC	100.0%	6.72	0.82
	BASE+PD	Duplicate.Assert	100.0%	7.21	0.96 (14.1%)
	BASE+PD+PR	Duplicate.Assert	50.29%	14.74	0.97 (17.7%)
		Duplicate.Assert.Added	49.71%		

Table A.6: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot χ^2	AUC (%)
Cxf	BASE	LOC	70.8%	217.57	0.74
		pre_bug	17.85%		
		deletedLine	3.51%		
		fileChurn	3.72%		
		ts_coupling	1.84%		
		tt_coupling	2.27%		
	BASE+PD	ts_coupling	9.09%	44.10	0.77 (3.4%)
		tt_coupling	11.19%		
		Assertion.Roulette	47.17%		
		Constructor.Initialization	12.22%		
		Exception.Catching.Throwing	20.34%		
	BASE+PD+PR	ts_coupling	5.97%	67.19	0.78 (5.0%)
		tt_coupling	7.34%		
		Assertion.Roulette	30.96%		
		Constructor.Initialization	8.02%		
		Exception.Catching.Throwing	13.35%		
		Sensitive.Equality.Added	10.22%		
		Exception.Catching.Throwing.Removed	8.44%		
		Eager.Test.Removed	9.34%		
		IgnoredTest.Removed	6.36%		
Groovy	BASE	LOC	100.0%	6.72	0.82
	BASE+PD	Duplicate.Assert	100.0%	7.21	0.96 (14.1%)
	BASE+PD+PR	Duplicate.Assert	50.29%	14.74	0.97 (17.7%)
		Duplicate.Assert.Added	49.71%		

Table A.7: The effect size of the test smell metrics on post-release defects. Effect is measured by setting the subject metric to 110 % and 150 % of it mean value, while other metrics are kept at their mean values. Bolded numbers indicate a positive increase in effect.

project	Metric Group	Individual Metric	110 % ↑	150 % ↑
Accumulo	TEST_PRODUCT	Conditional.Test.Logic	**0.12 %**	**0.62 %**
		Constructor.Initialization	-0.07 %	-0.34 %
		Redundant.Assertion	-0.04 %	-0.18 %
		Eager.Test	-0.4 %	-1.97 %
		Duplicate.Assert	-0.18 %	-0.91 %
	TEST_PROCESS	Eager.Test.Removed	-0.01 %	-0.04 %
		Duplicate.Assert.Added	**0.01 %**	**0.03 %**
		Unknown.Test.Added	-0.01 %	-0.03 %
Hive	TEST_PRODUCT	Conditional.Test.Logic	**0.08 %**	**0.41 %**
		EmptyTest	**0.01 %**	**0.03 %**
		General.Fixture	**0.05 %**	**0.27 %**
		Lazy.Test	-0.25 %	-1.22 %
		Duplicate.Assert	**0.05 %**	**0.26 %**
		Unknown.Test	**0.12 %**	**0.58 %**
		IgnoredTest	**0.01 %**	**0.06 %**
		Resource.Optimism	**0.07 %**	**0.33 %**
	TEST_PROCESS	Redundant.Assertion.Removed	-0.01 %	-0.01 %
		Sensitive.Equality.Removed	**0.01 %**	**0.04 %**
		Eager.Test.Removed	**0.01 %**	**0.01 %**
		Duplicate.Assert.Removed	**0.01 %**	**0.01 %**
		Unknown.Test.Removed	**0.06 %**	**0.29 %**
		Magic.Number.Test.Removed	-0.01 %	-0.06 %
		Conditional.Test.Logic.Added	-0.01 %	-0.01 %
		Mystery.Guest.Added	**0.01 %**	**0.03 %**
		Duplicate.Assert.Added	**0.01 %**	**0.04 %**
		IgnoredTest.Added	**0.01 %**	**0.03 %**
Wicket	TEST_PRODUCT	Conditional.Test.Logic	**0.01 %**	**0.01 %**
		Eager.Test	**0.01 %**	**0.03 %**
		Unknown.Test	**0.01 %**	**0.01 %**
	TEST_PROCESS	Assertion.Roulette.Added	-0.01 %	-0.01 %
		Conditional.Test.Logic.Added	-0.01 %	-0.01 %
		Exception.Catching.Throwing.Added	**0.01 %**	**0.01 %**
Cassandra	TEST_PROCESS	Conditional.Test.Logic.Removed	-0.01 %	-0.03 %
		Exception.Catching.Throwing.Removed	**0.01 %**	**0.07 %**
		IgnoredTest.Added	-0.01 %	-0.01 %
Bookkeeper	TEST_PRODUCT	Assertion.Roulette	-0.17 %	-0.82 %
		Constructor.Initialization	**0.33 %**	**1.81 %**
		Lazy.Test	-0.22 %	-1.01 %
		Unknown.Test	-0.09 %	-0.42 %
		Resource.Optimism	**0.07 %**	**0.36 %**
	TEST_PROCESS	General.Fixture.Removed	-0.01 %	-0.02 %
		Sleepy.Test.Removed	**0.01 %**	**0.02 %**
		Exception.Catching.Throwing.Added	**0.01 %**	**0.01 %**
		Lazy.Test.Added	-0.03 %	-0.14 %
		Unknown.Test.Added	-0.03 %	-0.17 %
		Magic.Number.Test.Added	-0.01 %	-0.01 %

Table A.8: The effect size of the test smell metrics on post-release defects. Effect is measured by setting the subject metric to 110 % and 150 % of it mean value, while other metrics are kept at their mean values. Bolded numbers indicate a positive increase in effect.

project	Metric Group	Individual Metric	110 % ↑	150 % ↑
Camel	TEST_PRODUCT	Assertion.Roulette	**0.01 %**	**0.01 %**
		Conditional.Test.Logic	-0.01 %	-0.01 %
		General.Fixture	-0.01 %	-0.01 %
		Mystery.Guest	**0.01 %**	**0.01 %**
		Redundant.Assertion	**0.01 %**	**0.01 %**
		Duplicate.Assert	**0.01 %**	**0.01 %**
		Resource.Optimism	**0.01 %**	**0.01 %**
	TEST_PROCESS	Print.Statement.Removed	**0.01 %**	**0.01 %**
		Eager.Test.Removed	-0.01 %	-0.01 %
		Print.Statement.Added	-0.01 %	-0.01 %
Groovy	TEST_PRODUCT	Duplicate.Assert	-0.01 %	-0.01 %
	TEST_PROCESS	Duplicate.Assert.Added	**0.01 %**	**0.01 %**
Karaf	TEST_PRODUCT	General.Fixture	**0.01 %**	**0.01 %**
		Print.Statement	**0.01 %**	**0.01 %**
		Sleepy.Test	**0.01 %**	**0.01 %**
		Unknown.Test	-0.01 %	-0.02 %
	TEST_PROCESS	General.Fixture.Removed	-0.01 %	-0.01 %
		Eager.Test.Added	**0.01 %**	**0.01 %**
		IgnoredTest.Added	-0.01 %	-0.01 %
		Magic.Number.Test.Added	**0.01 %**	**0.01 %**
Hadoop	TEST_PRODUCT	Duplicate.Assert	-0.01 %	-0.01 %
	TEST_PROCESS	Duplicate.Assert.Added	**0.01 %**	**0.01 %**

Table A.9: The effect size of the test smell metrics on post-release defects. Effect is measured by setting the subject metric to 110 % and 150 % of it mean value, while other metrics are kept at their mean values. Bolded numbers indicate a positive increase in effect.

project	Metric Group	Individual Metric	110 % ↑	150 % ↑
Cxf	TEST_PRODUCT	Assertion.Roulette	**0.01 %**	**0.08 %**
		Constructor.Initialization	-0.01 %	-0.03 %
		Exception.Catching.Throwing	**0.02 %**	**0.11 %**
	TEST_PROCESS	Exception.Catching.Throwing.Removed	**0.01 %**	**0.01 %**
		Eager.Test.Removed	-0.01 %	-0.02 %
		IgnoredTest.Removed	**0.01 %**	**0.01 %**
		Sensitive.Equality.Added	-0.01 %	-0.01 %
Flink	TEST_PRODUCT	Conditional.Test.Logic	**0.04 %**	**0.22 %**
		Exception.Catching.Throwing	-0.12 %	-0.57 %
		General.Fixture	**0.01 %**	**0.07 %**
		Mystery.Guest	**0.02 %**	**0.11 %**
		Lazy.Test	-0.07 %	-0.34 %
		Unknown.Test	-0.06 %	-0.31 %
		Magic.Number.Test	-0.05 %	-0.27 %
	TEST_PROCESS	Duplicate.Assert.Removed	-0.01 %	-0.02 %
		Magic.Number.Test.Removed	-0.01 %	-0.01 %
		EmptyTest.Added	-0.01 %	-0.02 %
		Mystery.Guest.Added	**0.01 %**	**0.01 %**
Zookeeper	TEST_PROCESS	Duplicate.Assert.Removed	-0.01 %	-0.01 %
		Unknown.Test.Removed	-0.01 %	-0.02 %
Kafka	TEST_PRODUCT	Exception.Catching.Throwing	**0.36 %**	**1.81 %**
		Mystery.Guest	**0.06 %**	**0.31 %**
		Redundant.Assertion	-0.04 %	-0.2 %
		Lazy.Test	-0.9 %	-4.32 %
		Duplicate.Assert	**0.17 %**	**0.84 %**
		Unknown.Test	-0.06 %	-0.28 %
		Magic.Number.Test	**0.33 %**	**1.66 %**
	TEST_PROCESS	Exception.Catching.Throwing.Removed	-0.01 %	-0.05 %
		Print.Statement.Removed	-0.03 %	-0.16 %
		Exception.Catching.Throwing.Added	**0.03 %**	**0.14 %**

References

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2017). Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*.

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2006). An evaluation of similarity coefficients for software fault localization. In *2006 12th pacific rim international symposium on dependable computing (prdc'06)* (pp. 39–46).

Akiyama, F. (1971). An example of software system debugging. In C. V. Freiman, J. E. Griffith, & J. L. Rosenfeld (Eds.), *Information processing, proceedings of IFIP, 1971* (pp. 353–359). North-Holland.

Akoglu, H. (2018). User's guide to correlation coefficients. *Turkish journal of emergency medicine*, *18*(3), 91–93.

AlDanial. (2019). *Count lines of code.* https://github.com/AlDanial/cloc. GitHub.

Ali, N. B., Engström, E., Taromirad, M., Mousavi, M. R., Minhas, N. M., Helgesson, D., ... Varshosaz, M. (2019). On the search for industry-relevant regression testing research. *Empirical Software Engineering*, *24*(4), 2020–2055. Retrieved from https://doi.org/10.1007/s10664-018-9670-1 doi: 10.1007/s10664-018-9670-1

Alipour, M. A., Shi, A., Gopinath, R., Marinov, D., & Groce, A. (2016). Evaluating non-adequate test-case reduction. In *Proceedings of the 31st ieee/acm international conference on automated software engineering* (pp. 16–26).

Almasi, M. M., Hemmati, H., Fraser, G., Arcuri, A., & Benefelds, J. (2017). An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 ieee/acm 39th international conference on software engineering: Software engineering*

in practice track (icse-seip) (pp. 263–272).

Alon, U., Brody, S., Levy, O., & Yahav, E. (2018). code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Alves, E. L., Song, M., & Kim, M. (2014). Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 751–754).

Apache. (2020). *Apache jenkins.* https://builds.apache.org/. (Last accessed April 3 2020)

apache dev. (2023). *https://lists.apache.org/list.html?dev@maven.apache.org.* Retrieved 2023-03-21, from https://lists.apache.org/thread/cpm046p745j7nj0dvw9mtxfmthkgobp6

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

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., & Binkley, D. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th ieee international conference on software maintenance (icsm)* (p. 56-65).

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., & Binkley, D. (2015). Are test smells really harmful? an empirical study. *Empirical Software Engineering*, *20*(4), 1052–1094.

Bavota, G., Qusef, A., Oliveto, R., Lucia, A. D., & Binkley, D. W. (2012a). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *28th IEEE international conference on software maintenance, ICSM* (pp. 56–65). IEEE Computer Society. Retrieved from https://doi.org/10.1109/ICSM.2012.6405253 doi: 10.1109/ICSM.2012.6405253

Bavota, G., Qusef, A., Oliveto, R., Lucia, A. D., & Binkley, D. W. (2012b). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *28th IEEE international conference on software maintenance, ICSM 2012, trento, italy, september 23-28, 2012* (pp. 56–65). IEEE Computer Society. Retrieved from https://doi.org/10.1109/ICSM.2012.6405253 doi:

10.1109/ICSM.2012.6405253

Beheshtian, M. J., Bavand, A. H., & Rigby, P. C. (2021). Software batch testing to save build test resources and to reduce feedback time. *IEEE Transactions on Software Engineering*, *48*(8), 2784–2801.

Bell, J., & Kaiser, G. (2014). Unit test virtualization with vmvm. In *Proceedings of the 36th international conference on software engineering* (pp. 550–561).

Biddle, R., & Tempero, E. (1996). Explaining inheritance: A code reusability perspective. In *Proceedings of the twenty-seventh sigcse technical symposium on computer science education* (pp. 217–221).

Bird, C., Nagappan, N., Murphy, B., Gall, H., & Devanbu, P. (2011). Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (p. 4-14).

Biyani, S., & Santhanam, P. (1998). Exploring defect data from development and customer usage on software modules over multiple releases. In *Ninth international symposium on software reliability engineering, ISSRE* (pp. 316–320). IEEE Computer Society. Retrieved from https://doi.org/10.1109/ISSRE.1998.730896 doi: 10.1109/ISSRE.1998.730896

Bleser, J. D., Nucci, D. D., & Roover, C. D. (2019). Assessing diffusion and perception of test smells in scala projects. In M. D. Storey, B. Adams, & S. Haiduc (Eds.), *Proceedings of the 16th international conference on mining software repositories, MSR 2019, 26-27 may 2019, montreal, canada* (pp. 457–467). IEEE / ACM. Retrieved from https://doi.org/10.1109/MSR.2019.00072 doi: 10.1109/MSR.2019.00072

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., ... others (2010). Managing technical debt in software-reliant systems. In *Proceedings of the fse/sdp workshop on future of software engineering research* (pp. 47–52).

Candido, J., Melo, L., & d'Amorim, M. (2017). Test suite parallelization in open-source projects: A study on its usage and impact. In *2017 32nd ieee/acm international conference on automated software engineering (ase)* (pp. 838–848).

Ceccato, M., Marchetto, A., Mariani, L., Nguyen, C. D., & Tonella, P. (2015). Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *25*(1), 1–38.

Chen, A. R., Chen, T.-H., & Wang, S. (2021). Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, *48*(8), 2905–2919.

Chen, J., Bai, Y., Hao, D., Xiong, Y., Zhang, H., & Xie, B. (2017). Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th international conference on software engineering* (pp. 700–711).

Chen, J., Bai, Y., Hao, D., Xiong, Y., Zhang, H., Zhang, L., & Xie, B. (2016). Test case prioritization for compilers: A text-vector based approach. In *2016 ieee international conference on software testing, verification and validation (icst)* (pp. 266–277).

Chen, T., Shang, W., Nagappan, M., Hassan, A. E., & Thomas, S. W. (2017). Topic-based software defect explanation. *J. Syst. Softw.*, *129*, 79–106. Retrieved from https://doi.org/10.1016/j.jss.2016.05.015 doi: 10.1016/j.jss.2016.05.015

Chen, T., Thomas, S. W., Hemmati, H., Nagappan, M., & Hassan, A. E. (2017, Sep.). An empirical study on the effect of testing on code quality using topic models: A case study on software development systems. *IEEE Transactions on Reliability*, *66*(3), 806-824.

Chen, T.-H., Shang, W., Hassan, A. E., Nasser, M., & Flora, P. (2016). Detecting problems in the database access code of large scale systems: An industrial experience report. In *Proceedings of the 38th international conference on software engineering* (p. 71–80).

Chen, T.-H., Thomas, S. W., Nagappan, M., & Hassan, A. (2012). Explaining software defects using topic models. In *Proceedings of the 9th working conference on mining software repositories.*

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), 476–493.

Child, M., Rosner, P., & Counsell, S. (2019). A comparison and evaluation of variants in

the coupling between objects metric. *J. Syst. Softw.*, *151*, 120–132. Retrieved from https://doi.org/10.1016/j.jss.2019.02.020 doi: 10.1016/j.jss.2019.02.020

Chowdhury, I., & Zulkernine, M. (2011). Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, *57*(3), 294–313.

cobertura. (2024). Retrieved from https://cobertura.github.io/cobertura/

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological measurement*, *20*(1), 37–46.

Cunningham, W. (1993). The wycash portfolio management system. *OOPS Messenger*, *4*(2), 29–30. Retrieved from https://doi.org/10.1145/157710.157715 doi: 10.1145/157710.157715

D'Ambros, M., Lanza, M., & Robbes, R. (2010). An extensive comparison of bug prediction approaches. In J. Whitehead & T. Zimmermann (Eds.), *Proceedings of the 7th international working conference on mining software repositories, MSR 2010 (co-located with icse), cape town, south africa, may 2-3, 2010, proceedings* (pp. 31–41). IEEE Computer Society. Retrieved from https://doi.org/10.1109/MSR.2010.5463279 doi: 10.1109/MSR.2010.5463279

Danglot, B., Vera-Perez, O., Yu, Z., Zaidman, A., Monperrus, M., & Baudry, B. (2019). A snowballing literature study on test amplification. *Journal of Systems and Software*, *157*, 110398.

de Pádua, G. B., & Shang, W. (2018). Studying the relationship between exception handling practices and post-release defects. In *Proceedings of the 15th international conference on mining software repositories, MSR 2018, gothenburg, sweden, may 28-29, 2018* (pp. 564–575).

Deursen, A., Moonen, L. M., Bergh, A., & Kok, G. (2001). *Refactoring test code* (Tech. Rep.). Amsterdam, The Netherlands, The Netherlands.

Dig, D., Manzoor, K., Johnson, R. E., & Nguyen, T. N. (2008). Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, *34*(3), 321–335.

Di Nardo, D., Alshahwan, N., Briand, L., & Labiche, Y. (2013). Coverage-based test case

prioritisation: An industrial case study. In *2013 ieee sixth international conference on software testing, verification and validation* (pp. 302–311).

Di Nardo, D., Alshahwan, N., Briand, L., & Labiche, Y. (2015). Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability*, *25*(4), 371–396.

Dyer, R., Rajan, H., Nguyen, H. A., & Nguyen, T. N. (2014). Mining billions of AST nodes to study actual and potential usage of java language features. In P. Jalote, L. C. Briand, & A. van der Hoek (Eds.), *36th international conference on software engineering, ICSE '14, hyderabad, india - may 31 - june 07, 2014* (pp. 779–790). ACM. Retrieved from https://doi.org/10.1145/2568225.2568295 doi: 10.1145/2568225.2568295

Eck, M., Palomba, F., Castelluccio, M., & Bacchelli, A. (2019). Understanding flaky tests: the developer's perspective. In M. Dumas, D. Pfahl, S. Apel, & A. Russo (Eds.), *Proceedings of the ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/SIGSOFT FSE 2019, tallinn, estonia, august 26-30, 2019* (pp. 830–840). ACM. Retrieved from https://doi.org/10.1145/3338906.3338945 doi: 10.1145/3338906.3338945

Empirical software engineering an international journal publishing model. (n.d.). Retrieved from https://link.springer.com/journal/10664

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., & Monperrus, M. (2014). Fine-grained and accurate source code differencing. In *Proceedings of the 29th acm/ieee international conference on automated software engineering* (pp. 313–324). New York, NY, USA: ACM. doi: 10.1145/2642937.2642982

Fang, Z. F., & Lam, P. (2015). Identifying test refactoring candidates with assertion fingerprints. In *Proceedings of the principles and practices of programming on the java platform* (pp. 125–137).

Fraser, G., & Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (pp. 416–419).

Fraser, G., & Arcuri, A. (2012). Whole test suite generation. *IEEE Transactions on Software Engineering*, *39*(2), 276–291.

Fraser, G., & Arcuri, A. (2015). 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering*, *20*, 611–639.

Garousi, V., & Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, *138*, 52–81. Retrieved from https://doi.org/10.1016/j.jss.2017.12.013 doi: 10.1016/j.jss.2017.12.013

Ge, X., Sarkar, S., Witschey, J., & Murphy-Hill, E. (2017). Refactoring-aware code review. In *2017 ieee symposium on visual languages and human-centric computing (vl/hcc)* (pp. 71–79).

Giordano, G., Fasulo, A., Catolino, G., Palomba, F., Ferrucci, F., & Gravino, C. (2022). On the evolution of inheritance and delegation mechanisms and their impact on code quality. In *2022 ieee international conference on software analysis, evolution and reengineering (saner)* (pp. 947–958).

Gligoric, M., Eloussi, L., & Marinov, D. (2015a). Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 international symposium on software testing and analysis* (pp. 211–222).

Gligoric, M., Eloussi, L., & Marinov, D. (2015b). Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 international symposium on software testing and analysis* (p. 211–222). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2771783.2771784 doi: 10.1145/2771783.2771784

Greiler, M., Van Deursen, A., & Storey, M.-A. (2013). Automated detection of test fixture strategies and smells. In *2013 ieee sixth international conference on software testing, verification and validation* (pp. 322–331).

Greiler, M., Zaidman, A., Van Deursen, A., & Storey, M.-A. (2013). Strategies for avoiding text fixture smells during software evolution. In *2013 10th working conference on mining software repositories (msr)* (pp. 387–396).

Groce, A., Alipour, M. A., Zhang, C., Chen, Y., & Regehr, J. (2014). Cause reduction for quick testing. In *2014 ieee seventh international conference on software testing, verification and validation* (pp. 243–252).

Grund, F., Chowdhury, S. A., Bradley, N., Hall, B., & Holmes, R. (2021). Codeshovel: Constructing method-level source code histories. In *Proceedings of the 43rd international conference on software engineering.*

Guo, Y., & Seaman, C. (2011). A portfolio approach to technical debt management. In *Proceedings of the 2nd workshop on managing technical debt* (pp. 31–34).

Gupta, N., Sharma, A., & Pachariya, M. K. (2019). An insight into test case optimization: ideas and trends with future perspectives. *IEEE Access*, *7*, 22310–22327.

Harrell Jr, F. E. (2015). *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis.* Springer.

Hossain, S. B., Dwyer, M. B., Elbaum, S., & Nguyen-Tuong, A. (2023a). Measuring and mitigating gaps in structural testing. In *2023 ieee/acm 45th international conference on software engineering (icse)* (pp. 1712–1723).

Hossain, S. B., Dwyer, M. B., Elbaum, S., & Nguyen-Tuong, A. (2023b). Measuring and mitigating gaps in structural testing. In *2023 ieee/acm 45th international conference on software engineering (icse)* (pp. 1712–1723).

Jacoco. (2003, March). *[java code coverage] reasons for huge performance impact.* Retrieved from https://jacoco.narkive.com/adoFZzfD/java-code-coverage-reasons-for-huge-performance-impact#:~:text=Usually%20the%20performance%20impact%20of,so%20a%20factor%20of%2010.

Jacoco. (2013). *Add parameter to include the current project in the aggregated report.* Retrieved 2023-06-26, from https://github.com/jacoco/jacoco/pull/1007

Jacoco. (2023). *Jacoco java code coverage library.* Retrieved from https://www.jacoco.org/jacoco/

Jagannath, V., Luo, Q., & Marinov, D. (2011). Change-aware preemption prioritization. In *Proceedings of the 2011 international symposium on software testing and analysis* (pp. 133–143).

Jia, Y., & Harman, M. (2009). Higher order mutation testing. *Information and Software Technology*, *51*(10), 1379–1393.

Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, *37*(5), 649-678. doi: 10.1109/TSE.2010.62

Jiarpakdee, J., Tantithamthavorn, C., & Hassan, A. E. (2018). The impact of correlated metrics on defect models. *CoRR*, *abs/1801.10271*. Retrieved from http://arxiv.org/abs/1801.10271

Jones, J. A., Harrold, M. J., & Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on software engineering* (pp. 467–477).

Junior, N. S., Soares, L. R., Martins, L. A., & Machado, I. (2020a). A survey on test practitioners' awareness of test smells. *CoRR*, *abs/2003.05613*. Retrieved from https://arxiv.org/abs/2003.05613

Junior, N. S., Soares, L. R., Martins, L. A., & Machado, I. (2020b). A survey on test practitioners' awareness of test smells. *CoRR*, *abs/2003.05613*. Retrieved from https://arxiv.org/abs/2003.05613

JUnit team. (2018, June). *Rules.* Retrieved from https://github.com/junit-team/junit4/wiki/Rules

Junit4. (2021, February). *Junit4.* Retrieved from https://junit.org/junit4/

Kainulainen, P. (2014). *Three reasons why we should not use inheritance in our tests.* Retrieved 2023-03-21, from https://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests/

Kang, S., Yoon, J., & Yoo, S. (2023). Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 ieee/acm 45th international conference on software engineering (icse)* (pp. 2312–2323).

Keller, F., Grunske, L., Heiden, S., Filieri, A., van Hoorn, A., & Lo, D. (2017). A critical evaluation of spectrum-based fault localization techniques on a large-scale software

system. In *2017 ieee international conference on software quality, reliability and security (qrs)* (pp. 114–125).

Ketkar, A., Tsantalis, N., & Dig, D. (2020a). Understanding type changes in java. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 629–641).

Ketkar, A., Tsantalis, N., & Dig, D. (2020b, Nov). Understanding type changes in java. In *Proceedings of the acm joint european software engineering conference and symposium on the foundations of software engineering.*

Khalek, S. A., & Khurshid, S. (2011). Efficiently running test suites using abstract undo operations. In *2011 ieee 22nd international symposium on software reliability engineering* (pp. 110–119).

Khomh, F., Di Penta, M., & Gueheneuc, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. In *2009 16th working conference on reverse engineering* (pp. 75–84).

Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, *17*, 243–275.

Kim, D. J., Chen, T.-H., & Yang, J. (2021a). The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, *26*, 1–47.

Kim, D. J., Chen, T.-H. P., & Yang, J. (2021b). The secret life of test smells - an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*.

Kim, D. J., Chen, T.-H. P., & Yang, J. (2024). A first look at the inheritance-induced redundant test execution.

Kim, D. J., Tsantalis, N., Chen, T.-H. P., & Yang, J. (2021). Studying test annotation maintenance in the wild. In (p. 62–73). IEEE Press. Retrieved from https://doi.org/10.1109/ICSE43902.2021.00019 doi: 10.1109/ICSE43902.2021.00019

Kim, D. J., Yang, B., Yang, J., & Chen, T.-H. (2021). How disabled tests manifest in test maintainability challenges? In *Proceedings of the 29th acm joint meeting*

on european software engineering conference and symposium on the foundations of software engineering (pp. 1045–1055).

Kim, J., Kim, J., & Yoo, S. (2017). Gpgpgpu: Evaluation of parallelisation of genetic programming using gpgpu. In *Search based software engineering: 9th international symposium, ssbse 2017, paderborn, germany, september 9-11, 2017, proceedings 9* (pp. 137–142).

Knuth, D. E. (1981). *Seminumerical algorithms* (2nd ed., Vol. 2). Reading, MA: Addison-Wesley.

Konsaard, P., & Ramingwong, L. (2015). Total coverage based regression test case prioritization using genetic algorithm. In *2015 12th international conference on electrical engineering/electronics, computer, telecommunications and information technology (ecti-con)* (pp. 1–6).

Kuhn, M., & Johnson, K. (2013). *Applied predictive modeling* (Vol. 26). Springer.

Lam, W., Godefroid, P., Nath, S., Santhiar, A., & Thummalapenta, S. (2019a). Root causing flaky tests in a large-scale industrial setting. In D. Zhang & A. Møller (Eds.), *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ISSTA* (pp. 101–111). ACM. Retrieved from https://doi.org/10.1145/3293882.3330570 doi: 10.1145/3293882.3330570

Lam, W., Godefroid, P., Nath, S., Santhiar, A., & Thummalapenta, S. (2019b). Root causing flaky tests in a large-scale industrial setting. In D. Zhang & A. Møller (Eds.), *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2019, beijing, china, july 15-19, 2019* (pp. 101–111). ACM. Retrieved from https://doi.org/10.1145/3293882.3330570 doi: 10.1145/3293882.3330570

Lam, W., Muslu, K., Sajnani, H., & Thummalapenta, S. (2020). A study on the lifecycle of flaky tests. In G. Rothermel & D. Bae (Eds.), *ICSE '20: 42nd international conference on software engineering, seoul, south korea, 27 june - 19 july, 2020* (pp. 1471–1482). ACM. Retrieved from https://doi.org/10.1145/3377811.3381749 doi: 10.1145/3377811.3381749

Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2019). idflakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)* (pp. 312–322).

Le, T.-D. B., Oentaryo, R. J., & Lo, D. (2015). Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 579–590).

Le, X.-B. D., Chu, D.-H., Lo, D., Le Goues, C., & Visser, W. (2017). Jfix: semantics-based repair of java programs via symbolic pathfinder. In *Proceedings of the 26th acm sigsoft international symposium on software testing and analysis* (pp. 376–379).

Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., & Marinov, D. (2016). An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (pp. 583–594).

Leitner, A., Oriol, M., Zeller, A., Ciupa, I., & Meyer, B. (2007). Efficient unit test case minimization. In *Proceedings of the 22nd ieee/acm international conference on automated software engineering* (pp. 417–420).

Li, Z., Harman, M., & Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, *33*(4), 225-237.

Liu, Z., Huang, Q., Xia, X., Shihab, E., Lo, D., & Li, S. (2018). SATD detector: a text-mining-based self-admitted technical debt detection tool. In M. Chaudron, I. Crnkovic, M. Chechik, & M. Harman (Eds.), *Proceedings of the 40th international conference on software engineering: Companion proceeedings, ICSE 2018, gothenburg, sweden, may 27 - june 03, 2018* (pp. 9–12). ACM. Retrieved from https://doi.org/10.1145/3183440.3183478 doi: 10.1145/3183440.3183478

Lucas, W., Bonifácio, R., Canedo, E. D., Marcílio, D., & Lima, F. (2019). Does the introduction of lambda expressions improve the comprehension of java programs? In *Proceedings of the xxxiii brazilian symposium on software engineering* (pp. 187–196).

Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014a). An empirical analysis of flaky tests. In S. Cheung, A. Orso, & M. D. Storey (Eds.), *Proceedings of the 22nd ACM*

SIGSOFT international symposium on foundations of software engineering, (fse-22), hong kong, china, november 16 - 22, 2014 (pp. 643–653). ACM. Retrieved from https://doi.org/10.1145/2635868.2635920 doi: 10.1145/2635868.2635920

Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014b). An empirical analysis of flaky tests. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 643–653).

Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014c). An empirical analysis of flaky tests. In S. Cheung, A. Orso, & M. D. Storey (Eds.), *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, (fse-22), hong kong, china, november 16 - 22, 2014* (pp. 643–653). ACM. Retrieved from https://doi.org/10.1145/2635868.2635920 doi: 10.1145/2635868.2635920

Marinescu, C., & Codoban, M. (2014). Should we beware the inheritance? an empirical study on the evolution of seven open source systems. In *2014 9th international conference on software engineering and applications (icsoft-ea)* (pp. 246–253).

Marri, M. R., Tao Xie, Tillmann, N., de Halleux, J., & Schulte, W. (2009). An empirical study of testing file-system-dependent software with mock objects. In *2009 icse workshop on automation of software test* (p. 149-153).

Marsavina, C., Romano, D., & Zaidman, A. (2014). Studying fine-grained co-evolution patterns of production and test code. In *2014 ieee 14th international working conference on source code analysis and manipulation* (pp. 195–204).

McMaster, S., & Memon, A. (2007). Fault detection probability analysis for coverage-based test suite reduction. In *2007 ieee international conference on software maintenance* (pp. 335–344).

Mechtaev, S., Yi, J., & Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering* (pp. 691–701).

Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., & Rothermel, G. (2012, November). A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, *38*(6), 1258–1275.

Mens, T., Demeyer, S., D'Ambros, M., Gall, H., Lanza, M., & Pinzger, M. (2008). Analysing software repositories to understand software evolution. *Software evolution*, 37–67.

Meszaros, G. (2007). *xunit test patterns: Refactoring test code*. Pearson Education.

Moser, R., Pedrycz, W., & Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In W. Schäfer, M. B. Dwyer, & V. Gruhn (Eds.), *30th international conference on software engineering (ICSE 2008), leipzig, germany, may 10-18, 2008* (pp. 181–190). ACM. Retrieved from https://doi.org/10.1145/1368088.1368114 doi: 10.1145/1368088.1368114

Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G. (2006). Does refactoring improve reusability? In *International conference on software reuse* (pp. 287–297).

Nadeem, A., Awais, A., et al. (2006). Testfilter: a statement-coverage based test case reduction technique. In *2006 ieee international multitopic conference* (pp. 275–280).

Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on software engineering* (pp. 284–292).

Nagappan, N., Ball, T., & Zeller, A. (2006). Mining metrics to predict component failures. In L. J. Osterweil, H. D. Rombach, & M. L. Soffa (Eds.), *28th international conference on software engineering (ICSE 2006), shanghai, china, may 20-28, 2006* (pp. 452–461). ACM. Retrieved from https://doi.org/10.1145/1134285.1134349 doi: 10.1145/1134285.1134349

Nasseri, E., Counsell, S., & Shepperd, M. (2008). An empirical study of evolution of inheritance in java oss. In *19th australian conference on software engineering (aswec 2008)* (pp. 269–278).

Nasseri, E., Counsell, S., & Shepperd, M. (2010). Class movement and re-location: An empirical study of java inheritance evolution. *Journal of Systems and Software*, *83*(2), 303–315.

Nirpal, P. B., & Kale, K. (2011). Using genetic algorithm for automated efficient software test case generation for path testing. *International Journal of Advanced Networking and Applications*, *2*(6), 911–915.

Oracle. (2022a). *Inheritance.* Retrieved 2023-03-21, from https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

Oracle. (2022b). *Polymorphism.* Retrieved 2023-03-21, from https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

Pacheco, C., & Ernst, M. D. (2007). Randoop: feedback-directed random testing for java. In *Companion to the 22nd acm sigplan conference on object-oriented programming systems and applications companion* (pp. 815–816).

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th international conference on software engineering* (pp. 482–482).

Pan, R., Ghaleb, T. A., & Briand, L. (2023). Atm: Black-box test case minimization based on test code similarity and evolutionary search. In *2023 ieee/acm 45th international conference on software engineering (icse)* (pp. 1700–1711).

Panichella, A., Panichella, S., Fraser, G., Sawant, A. A., & Hellendoorn, V. J. (2022). Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering*, *27*(7), 170.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., & Harman, M. (2019). Mutation testing advances: an analysis and survey. In *Advances in computers* (Vol. 112, pp. 275–378). Elsevier.

Parnin, C., Bird, C., & Murphy-Hill, E. R. (2013). Adoption and use of java generics. *Empirical Software Engineering*, *18*(6), 1047–1089. Retrieved from https://doi.org/10.1007/s10664-012-9236-6 doi: 10.1007/s10664-012-9236-6

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., & Seinturier, L. (2015). Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, *46*, 1155-1179. Retrieved from https://hal.archives-ouvertes.fr/hal-01078532/document doi: 10.1002/spe.2346

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., & Seinturier, L. (2016). Spoon: A library for implementing analyses and transformations of java source code. *Software:*

Practice and Experience, *46*(9), 1155–1179.

Peng, Z., Chen, T.-H., & Yang, J. (2020a). Revisiting test impact analysis in continuous testing from the perspective of code dependencies. *IEEE Transactions on Software Engineering*, 1-1. doi: 10.1109/TSE.2020.3045914

Peng, Z., Chen, T.-H., & Yang, J. (2020b). Revisiting test impact analysis in continuous testing from the perspective of code dependencies. *IEEE Transactions on Software Engineering*.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., & Palomba, F. (2019a). On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th annual international conference on computer science and software engineering* (pp. 193–202).

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., & Palomba, F. (2019b). On the distribution of test smells in open source android applications: an exploratory study. In T. Pakfetrat, G. Jourdan, K. Kontogiannis, & R. F. Enenkel (Eds.), *Proceedings of the 29th annual international conference on computer science and software engineering, CASCON 2019, markham, ontario, canada, november 4-6, 2019* (pp. 193–202). ACM. Retrieved from https://dl.acm.org/doi/abs/10.5555/3370272.3370293

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., & Palomba, F. (2020, Nov.). tsdetect: An open source test smells detection tool. In *Proceedings of the 2020 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering.* New York, NY, USA: Association for Computing Machinery. Retrieved from https://testsmells.github.io/assets/publications/FSE2020_TechnicalPaper.pdf doi: 10.1145/3368089.3417921

Pham, T. M.-T., & Yang, J. (2020a). The secret life of commented-out source code. In *28th ieee/acm international conference on program comprehension, ICSE.*

Pham, T. M. T., & Yang, J. (2020b). The secret life of commented-out source code. In *ICPC '20: 28th international conference on program comprehension, seoul, republic of korea, july 13-15, 2020* (pp. 308–318). ACM. Retrieved from https://doi.org/

10.1145/3387904.3389259 doi: 10.1145/3387904.3389259

Pinto, L. S., Sinha, S., & Orso, A. (2012a). Understanding myths and realities of test-suite evolution. In W. Tracz, M. P. Robillard, & T. Bultan (Eds.), *20th ACM SIGSOFT symposium on the foundations of software engineering (fse-20), sigsoft/fse'12, cary, nc, USA - november 11 - 16, 2012* (p. 33). ACM. Retrieved from https://doi.org/10.1145/2393596.2393634 doi: 10.1145/2393596.2393634

Pinto, L. S., Sinha, S., & Orso, A. (2012b). Understanding myths and realities of test-suite evolution. In W. Tracz, M. P. Robillard, & T. Bultan (Eds.), *20th ACM SIGSOFT symposium on the foundations of software engineering (fse-20), sigsoft/fse'12, cary, nc, USA - november 11 - 16, 2012* (p. 33). ACM. Retrieved from https://doi.org/10.1145/2393596.2393634 doi: 10.1145/2393596.2393634

Piotrowski, P., & Madeyski, L. (2020). Software defect prediction using bad code smells: A systematic literature review. In *Data-centric business and applications* (pp. 77–99). Retrieved from https://doi.org/10.1007/978-3-030-34706-2_5 doi: 10.1007/978-3-030-34706-2_5

Potdar, A., & Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *2014 ieee international conference on software maintenance and evolution* (pp. 91–100).

Prechelt, L., Unger, B., Philippsen, M., & Tichy, W. (2003). A controlled experiment on inheritance depth as a cost factor for code maintenance. *Journal of Systems and Software*, *65*(2), 115–126.

Programming, E. E. (2011, March). *3a – arrange, act, assert.* Retrieved from https://xp123.com/3a-arrange-act-assert/

Qusef, A., Elish, M. O., & Binkley, D. W. (2019). An exploratory study of the relationship between software test smells and fault-proneness. *IEEE Access*, *7*, 139526–139536. Retrieved from https://doi.org/10.1109/ACCESS.2019.2943488 doi: 10.1109/ACCESS.2019.2943488

Rahman, F., & Devanbu, P. T. (2011). Ownership, experience and defects: a fine-grained study of authorship. In R. N. Taylor, H. C. Gall, & N. Medvidovic (Eds.), *Proceedings*

of the 33rd international conference on software engineering, ICSE 2011, waikiki, honolulu , hi, usa, may 21-28, 2011 (pp. 491–500). ACM. Retrieved from https://doi.org/10.1145/1985793.1985860 doi: 10.1145/1985793.1985860

Rocha, H., & Valente, M. T. (2011). How annotations are used in java: An empirical study. In *Proceedings of the 23rd international conference on software engineering & knowledge engineering (seke'2011), eden roc renaissance, miami beach, usa, july 7-9, 2011* (pp. 426–431). Knowledge Systems Institute Graduate School.

Rothermel, G., & Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology*, *6*(2), 173–210.

Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, *27*(10), 929-948.

Saha, R. K., Zhang, L., Khurshid, S., & Perry, D. E. (2015). An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the 37th international conference on software engineering* (pp. 268–279).

Shaheen, M. R., & du Bousquet, L. (2008). Relation between depth of inheritance tree and number of methods to test. In *2008 1st international conference on software testing, verification, and validation* (pp. 161–170).

Shang, W., Nagappan, M., & Hassan, A. E. (2015). Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, *20*(1), 1–27.

Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., & Wang, Q. (2019). Intellimerge: A refactoring-aware software merging technique. *Proceedings of the ACM on Programming Languages*, *3*(OOPSLA), 1–28.

Shi, A., Bell, J., & Marinov, D. (2019). Mitigating the effects of flaky tests on mutation testing. In D. Zhang & A. Møller (Eds.), *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ISSTA* (pp. 112–122). ACM. Retrieved from https://doi.org/10.1145/3293882.3330568 doi: 10.1145/3293882.3330568

Shi, A., Yung, T., Gyori, A., & Marinov, D. (2015). Comparing and combining test-suite

reduction and regression test selection. In *Proceedings of the 10th joint meeting on foundations of software engineering* (pp. 237–247).

Siebra, C. S. A., Tonin, G. S., Silva, F. Q., Oliveira, R. G., Junior, A. L., Miranda, R. C., & Santos, A. L. (2012). Managing technical debt in practice: An industrial report. In *Proceedings of the acm-ieee international symposium on empirical software engineering and measurement* (pp. 247–250).

Silva, D., Tsantalis, N., & Valente, M. T. (2016). Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (pp. 858–870).

Spadini, D., Aniche, M., Bruntink, M., & Bacchelli, A. (2019, June). Mock objects for testing java systems. , *24*(3), 1461–1498.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., & Bacchelli, A. (2018). On the relation of test smells to software code quality. In *2018 ieee international conference on software maintenance and evolution (icsme)* (p. 1-12).

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., & Bacchelli, A. (2018). On the relation of test smells to software code quality. In *2018 IEEE international conference on software maintenance and evolution, ICSME 2018, madrid, spain, september 23-29, 2018* (pp. 1–12). IEEE Computer Society. Retrieved from https://doi.org/10.1109/ICSME.2018.00010 doi: 10.1109/ICSME.2018.00010

Spadini, D., Schvarcbacher, M., Oprescu, A., Bruntink, M., & Bacchelli, A. (2020). Investigating severity thresholds for test smells. In S. Kim, G. Gousios, S. Nadi, & J. Hejderup (Eds.), *MSR '20: 17th international conference on mining software repositories, seoul, republic of korea, 29-30 june, 2020* (pp. 311–321). ACM. Retrieved from https://doi.org/10.1145/3379597.3387453 doi: 10.1145/3379597.3387453

Spinellis, D., Louridas, P., & Kechagia, M. (2021). Software evolution: the lifetime of fine-grained elements. *PeerJ Computer Science*, *7*, e372.

stackoverflow. (2013a). *Prefer composition over inheritance?* Retrieved 2023-06-26, from https://stackoverflow.com/questions/49002/prefer-composition-over-inheritance

stackoverflow. (2013b). *why inheritance is strongly coupled where as composition is loosely coupled in java?* Retrieved 2023-03-21, from https://stackoverflow.com/questions/19146979/why-inheritance-is-strongly-coupled-where-as-composition-is-loosely-coupled-in-j

stackoverflow. (2020). *How can i resolve this redundancy caused by inheritance and nested class?* Retrieved 2023-03-21, from https://stackoverflow.com/questions/56063575/how-can-i-resolve-this-redundancy-caused-by-inheritance-and-nested-class

Stackoverflow. (2023). *Should i use inherited tests?* Retrieved 2023-03-21, from https://stackoverflow.com/questions/59312507/should-i-use-inherited-tests

Subramanyam, R., & Krishnan, M. S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, *29*(4), 297–310.

Tahir, A., Counsell, S., & MacDonell, S. G. (2016). *An empirical study into the relationship between class features and test smells. in 2016 23rd asia-pacific software engineering conference (apsec).* IEEE.

Tan, L., Yuan, D., Krishna, G., & Zhou, Y. (2007). /*icomment: bugs or bad comments?*/. In T. C. Bressoud & M. F. Kaashoek (Eds.), *Proceedings of the 21st ACM symposium on operating systems principles 2007, SOSP 2007, stevenson, washington, usa, october 14-17, 2007* (pp. 145–158). ACM. Retrieved from https://doi.org/10.1145/1294261.1294276 doi: 10.1145/1294261.1294276

TestNG. (2023, May). *Testng.* Retrieved from https://testng.org/annotations.html

Thomas, S. W., Hemmati, H., Hassan, A. E., & Blostein, D. (2014, February). Static test case prioritization using topic models. *Empirical Software Engineering*, *19*(1), 182–212.

Tomassetti, F., Smith, N., Maximilien, C., & Kirsch, S. (2021). *Javaparser.*

Tsantalis, N., Ketkar, A., & Dig, D. (2020a). Refactoringminer 2.0. *IEEE Transactions on Software Engineering*.

Tsantalis, N., Ketkar, A., & Dig, D. (2020b). Refactoringminer 2.0. *IEEE Transactions on*

Software Engineering. doi: 10.1109/TSE.2020.3007722

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., & Dig, D. (2018a). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering* (pp. 483–494). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3180155.3180206 doi: 10.1145/3180155.3180206

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., & Dig, D. (2018b). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering* (pp. 483–494). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3180155.3180206 doi: 10.1145/3180155.3180206

Tufano, M., Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., & Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering* (pp. 4–15).

Vahabzadeh, A., Stocco, A., & Mesbah, A. (2018). Fine-grained test minimization. In *2018 ieee/acm 40th international conference on software engineering (icse)* (pp. 210–221).

Van Deursen, A., Moonen, L., Van Den Bergh, A., & Kok, G. (2001). Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (xp2001)* (pp. 92–95).

Viera, A. J., Garrett, J. M., et al. (2005). Understanding interobserver agreement: the kappa statistic. *Family Medicine*, *37*(5), 360–363.

Wang, K., Zhu, C., Celik, A., Kim, J., Batory, D., & Gligoric, M. (2018). Towards refactoring-aware regression test selection. In *Proceedings of the 40th international conference on software engineering* (pp. 233–244).

Wang, S., Chen, T.-H., & Hassan, A. E. (2018). Understanding the factors for fast answers in technical q&a websites. *Empirical Software Engineering*, *23*(3), 1552–1593.

Wang, S., Nam, J., & Tan, L. (2017). Qtep: Quality-aware test case prioritization. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*

(pp. 523–534).

Wang, X., Xiao, L., Yu, T., Woepse, A., & Wong, S. (2021). An automatic refactoring framework for replacing test-production inheritance by mocking mechanism. In *Proceedings of the 29th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 540–552).

Wehaibi, S., Shihab, E., & Guerrouj, L. (2016). Examining the impact of self-admitted technical debt on software quality. In *IEEE 23rd international conference on software analysis, evolution, and reengineering, SANER 2016, suita, osaka, japan, march 14-18, 2016 - volume 1* (pp. 179–188). IEEE Computer Society. Retrieved from https://doi.org/10.1109/SANER.2016.72 doi: 10.1109/SANER.2016.72

Wikipedia. (2023a, March). *Dijkstra's algorithm.* Retrieved from https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Wikipedia. (2023b, March). *Disjoint-set data structure.* Retrieved from https://en.wikipedia.org/wiki/Disjoint-set_data_structure

Wiklund, K., Eldh, S., Sundmark, D., & Lundqvist, K. (2012). Technical debt in test automation. In *2012 ieee fifth international conference on software testing, verification and validation* (pp. 887–892).

Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, *42*(8), 707–740.

Ye, H., Martinez, M., Durieux, T., & Monperrus, M. (2021). A comprehensive study of automatic program repair on the quixbugs benchmark. *Journal of Systems and Software*, *171*, 110825.

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, *22*(2), 67–120.

Yoo, S., Harman, M., & Clark, D. (2013, July). Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering Methodology*, *22*(3), 19:1–19:29.

Yu, C. S., Treude, C., & Aniche, M. F. (2019). Comprehending test code: An empirical study. *CoRR*, *abs/1907.13365*. Retrieved from http://arxiv.org/abs/1907.13365

Yu, Z., Bai, C., Seinturier, L., & Monperrus, M. (2019). Characterizing the usage, evolution and impact of java annotations in practice. *IEEE Transactions on Software Engineering*, *47*(5), 969–986.

Zaidman, A., Rompaey, B. V., Demeyer, S., & van Deursen, A. (2008). Mining software repositories to study co-evolution of production test code. In *2008 1st international conference on software testing, verification, and validation* (p. 220-229).

Zaidman, A., Van Rompaey, B., Demeyer, S., & Van Deursen, A. (2008). Mining software repositories to study co-evolution of production & test code. In *2008 1st international conference on software testing, verification, and validation* (pp. 220–229).

Zaidman, A., Van Rompaey, B., Van Deursen, A., & Demeyer, S. (2011). Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, *16*, 325–364.

Zeller, A. (2009). *Why programs fail - A guide to systematic debugging, 2nd edition.* Academic Press. Retrieved from http://store.elsevier.com/product.jsp?isbn=9780123745156&pagename=search

Zerouali, A., & Mens, T. (2017). Analyzing the evolution of testing library usage in open source java projects. In *2017 ieee 24th international conference on software analysis, evolution and reengineering (saner)* (p. 417-421).

Zerouali, A., & Mens, T. (2017). Analyzing the evolution of testing library usage in open source java projects. In M. Pinzger, G. Bavota, & A. Marcus (Eds.), *IEEE 24th international conference on software analysis, evolution and reengineering, SANER 2017, klagenfurt, austria, february 20-24, 2017* (pp. 417–421). IEEE Computer Society. Retrieved from https://doi.org/10.1109/SANER.2017.7884645 doi: 10.1109/SANER.2017.7884645

Zhang, L., Hao, D., Zhang, L., Rothermel, G., & Mei, H. (2013). Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 international conference on software engineering* (pp. 192–201).

Zhang, L., Marinov, D., Zhang, L., & Khurshid, S. (2011). An empirical study of junit test-suite reduction. In *Proceedings of the ieee 22nd international symposium on software*

reliability engineering (pp. 170–179).

Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M. D., & Notkin, D. (2014). Empirically revisiting the test independence assumption. In (p. 385–396). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2610384.2610404 doi: 10.1145/2610384.2610404

Zhao, X., Liang, J., & Dang, C. (2019). A stratified sampling based clustering algorithm for large-scale data. *Knowl. Based Syst.*, *163*, 416–428. Retrieved from https://doi.org/10.1016/j.knosys.2018.09.007 doi: 10.1016/j.knosys.2018.09.007

Zimmermann, T., Nagappan, N., & Williams, L. (2010). Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 third international conference on software testing, verification and validation* (pp. 421–428).

Zimmermann, T., Premraj, R., & Zeller, A. (2007). Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering* (p. 9).