Delft University of Technology

An Empirical Evaluation of Feedback-Driven Software Development

Beller, Moritz

DOI
10.4233/uuid:b2946104-2092-42bb-a1ee-3b085d110466

Publication date
2018

Document Version
Final published version

Citation (APA)
Beller, M. (2018). *An Empirical Evaluation of Feedback-Driven Software Development*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:b2946104-2092-42bb-a1ee-3b085d110466

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

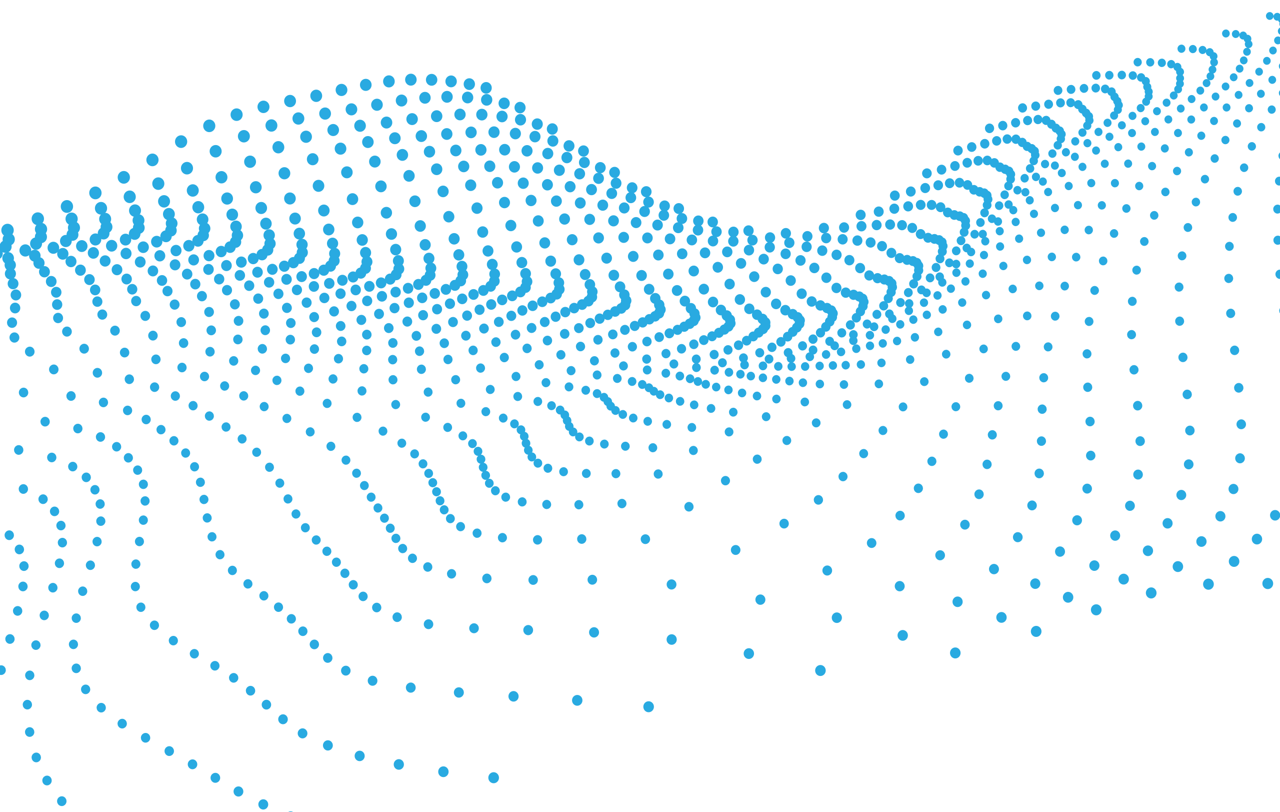
Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

AN EMPIRICAL EVALUATION OF FEEDBACK-DRIVEN SOFTWARE DEVELOPMENT

Moritz Beller



An Empirical Evaluation of Feedback-Driven Software Development

An Empirical Evaluation of Feedback-Driven Software Development

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op vrijdag 23 november 2018 om 15.00 uur

door

Moritz Marc BELLER

Master of Science in Computer Science,
Technische Universität München, Duitsland,
geboren te Schweinfurt, Duitsland.

Dit proefschrift is goedgekeurd door de

promotoren: Dr. A.E. Zaidman, Prof. dr. A. van Deursen
copromotor: Dr. ir. G. Gousios

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. A. van Deursen,	Technische Universiteit Delft
Dr. A.E. Zaidman,	Technische Universiteit Delft
Dr. ir. G. Gousios,	Technische Universiteit Delft
Onafhankelijke leden:	
Prof. dr. ir. G.J.P.M. Houben,	Technische Universiteit Delft
Prof. dr. P. Runeson,	Lund Universitet, Sweden
Dr. Th. Zimmermann,	Microsoft Research, United States of America
Prof. dr. D. Spinellis,	Athens University of Economics and Business, Greece
Prof. dr. ir. E. Visser,	Technische Universiteit Delft, reservelid

Prof. dr. D. Spinellis has contributed to the end phase of writing Chapter 6.

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and was financed by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO), project TestRoots, grant number 016.133.324.

Keywords: Feedback-Driven Development (FDD), Developer Testing, Empirical Software Engineering, Continuous Integration

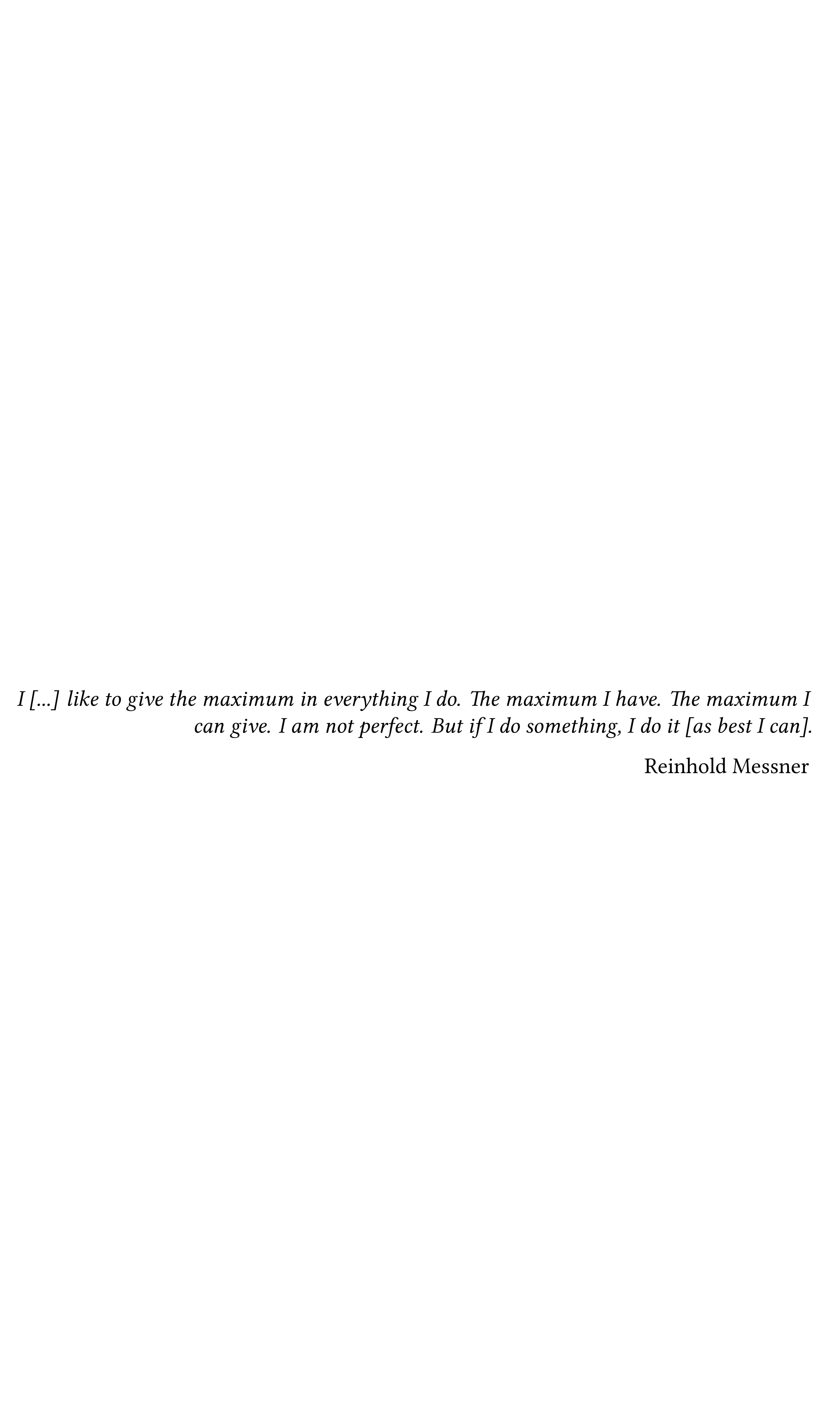
Printed by: ProefschriftMaken, `www.proefschriftmaken.nl`

Cover: Cloud of ‘2,443 points’ by Zsófia Varga

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

ISBN 978-94-6380-065-5

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.



I [...] like to give the maximum in everything I do. The maximum I have. The maximum I can give. I am not perfect. But if I do something, I do it [as best I can].

Reinhold Messner

Contents

Summary **xi**

Samenvatting **xiii**

Acknowledgments **xv**

1 Introduction **1**

1.1 Background & Context . . . 2
1.1.1 A Model of Feedback-Driven Development . . . 2
1.1.2 The Case for FDD in a Collaborative Coding World . . . 5
1.2 Feedback-Driven Development in Practice . . . 6
1.3 Research Goal and Questions . . . 8
1.4 Research Methodology . . . 8
1.4.1 Research Method Categorization . . . 9
1.4.2 Enablement of Large-Scale Studies . . . 10
1.4.3 Ethical Implications . . . 11
1.5 Replicability, Open Science & Source . . . 12
1.5.1 Open Data Sets . . . 12
1.5.2 Open-Source Contributions . . . 13
1.6 Outline & Contribution . . . 14
1.6.1 Thesis Structure . . . 16
1.6.2 Other Contributions . . . 18

2 Analyzing the State of Static Analysis **21**

2.1 Related Work . . . 23
2.1.1 Automatic Static Analysis Tools . . . 23
2.1.2 Defect Classifications . . . 23
2.2 Research Questions . . . 24
2.3 Prevalence Analysis (RQ I.1) . . . 25
2.3.1 Methodology . . . 25
2.3.2 Results . . . 26
2.4 General Defect Classification (GDC) . . . 27
2.5 Configuration & Evolution (RQ I.2, RQ I.3) . . . 28
2.5.1 Study Design . . . 29
2.5.2 Methods . . . 29
2.5.3 Study Objects . . . 31
2.5.4 Results . . . 32
2.6 Discussion . . . 36
2.6.1 Results . . . 36
2.6.2 Threats to Validity . . . 39

2.7 Tool Construction UAV . . . 40
2.7.1 Introduction . . . 40
2.7.2 User Story . . . 41
2.7.3 Related Work . . . 41
2.7.4 Implementation . . . 43
2.7.5 Evaluation . . . 46
2.7.6 Development Roadmap . . . 47
2.8 Future Work & Conclusions . . . 48

3 The Last Line Effect Explained 51
3.1 Study Setup . . . 54
3.1.1 Study Design C_1: Spread and Prevalence of the Last Line Effect within Micro-Clones . . . 54
3.1.2 Study Design C_2: Analyzing Reasons Behind the Existence of the Last Line Effect . . . 55
3.1.3 Study Objects . . . 56
3.1.4 How to Replicate This Study . . . 56
3.2 Methods . . . 56
3.2.1 Inaptness of Current Clone Detectors . . . 56
3.2.2 How to Find Faulty Micro-Clones Instead . . . 57
3.2.3 Inferring the Origin of an Erroneous Micro-Clone Instance . . . 57
3.2.4 Putting Commit Sizes in Perspective . . . 59
3.3 Results . . . 59
3.3.1 Overview Description of Results . . . 59
3.3.2 In-Depth Investigation of Findings . . . 60
3.3.3 Statistical Evaluation . . . 63
3.3.4 Origin of Micro-Clones . . . 64
3.3.5 Developer Interviews . . . 66
3.3.6 Usefulness of Results . . . 69
3.4 Discussion . . . 69
3.4.1 Technical Complexity & Reasons . . . 70
3.4.2 Psychological Mechanisms & Reasons . . . 70
3.4.3 Threats to Validity . . . 72
3.5 Related Work . . . 74
3.6 Future Work & Conclusion . . . 75

4 Developer Testing in the IDE: Patterns, Beliefs, and Behavior 77
4.1 Study Infrastructure Design . . . 79
4.1.1 Field Study Infrastructure . . . 79
4.1.2 WatchDog Developer Survey & Testing Analytics . . . 81
4.1.3 IDE Instrumentation . . . 84
4.2 Research Methods . . . 88
4.2.1 Correlation Analyses (RQ III.1, RQ III.2) . . . 88
4.2.2 Analysis of Induced Test Failures (RQ III.3) . . . 88
4.2.3 Sequentialization of Intervals (RQ III.3, RQ III.4) . . . 89
4.2.4 Test Flakiness Detection (RQ III.3) . . . 89

4.2.5 Recognition of Test-Driven Development (RQ III.4). 89
4.2.6 Statistical Evaluation (RQ III.1–RQ III.5) 92
4.3 Study Participants 92
4.3.1 Acquisition of Participants 92
4.3.2 Demographics of Study Subjects 93
4.3.3 Data Normalization 95
4.4 Results 95
4.4.1 RQ III.1: Which Testing Patterns Are Common In the IDE?. 95
4.4.2 RQ III.2: What Characterizes The Tests Developers Run In The IDE? 97
4.4.3 RQ III.3: How Do Developers Manage Failing Tests? 98
4.4.4 RQ III.4: Do Developers Follow TDD In The IDE? 101
4.4.5 RQ III.5: How Much Do Developers Test In The IDE?. 102
4.5 Discussion 103
4.5.1 RQ III.1: Which Testing Patterns Are Common In the IDE?. 104
4.5.2 RQ III.2: What Characterizes The Tests Developers Run? 105
4.5.3 RQ III.3: How Do Developers Manage Failing Tests? 106
4.5.4 RQ III.4: Do Developers Follow TDD? 108
4.5.5 RQ III.5: How Much Do Developers Test? 110
4.5.6 A Note On Generality And Replicability 112
4.5.7 Toward A theory of Test-Guided Development 112
4.6 Threats to Validity 113
4.6.1 Limitations 113
4.6.2 Construct Validity 114
4.6.3 Internal Validity 114
4.6.4 External Validity 115
4.7 Related Work. 116
4.7.1 Related Tools and Plugins 116
4.7.2 Related Research 116
4.8 Conclusion 117

5 Oops, My Tests Broke the Build: An Analysis of Travis CI 119
5.1 Background 122
5.1.1 Related Work. 122
5.1.2 Travis CI 122
5.2 Research Setup 125
5.2.1 Study Design 125
5.2.2 Tools 125
5.2.3 Build Linearization and Mapping to Git 127
5.2.4 Statistical Evaluation 129
5.3 The TravisTorrent Data Set 129
5.3.1 Descriptive Statistics 129
5.3.2 Data-Set-as-a-Service 129
5.3.3 Data Sample 130

5.4 Results 132
5.4.1 RQ IV.1: How common is Travis CI use on GitHub? 132
5.4.2 RQ IV.2: How central is testing to CI? 134
5.4.3 RQ IV.3: How do tests influence the build result? 137
5.5 Discussion 139
5.5.1 Results 139
5.5.2 Threats to Validity 142
5.6 Future Work 143
5.7 Conclusion 144

6 On the Dichotomy of Debugging Behavior Among Programmers 145
6.1 Related Work 147
6.2 Debugging Survey 149
6.2.1 Research Methods 149
6.2.2 Results 150
6.3 IDE Field Study 154
6.3.1 Study Methods 154
6.3.2 Results 155
6.4 Interviews 160
6.4.1 Study Methods 160
6.4.2 Results 160
6.5 Threats to Validity 164
6.6 Conclusion 165

7 Conclusion 167
7.1 Research Questions Revisited 167
7.2 Threats to Validity 169
7.3 A Speculative Perspective on Feedback-Driven Development 170
7.4 Implications 172
7.4.1 Individual FDD Stages 172
7.4.2 Conclusion and Future Work on FDD 174

Bibliography 175

Glossary 205

Curriculum Vitæ 207

List of Publications 211

Summary

Software developers today crave for feedback, be it from their peers in the form of code review, static analysis tools like their compiler, or the local or remote execution of their tests in the Continuous Integration (CI) environment. With the advent of social coding sites such as GitHub and tight integration of CI services such as Travis CI, software development practices have fundamentally changed. Despite a highly alternated software engineering landscape, however, we still lack a suitable holistic description of contemporary software development practices. Existing descriptions such as the V-model are either too coarse-grained to describe an individual contributor's workflow, or only regard a subpart of the development process, like Test-Driven Development (TDD). In addition, most existing models are *pre*- rather than *de*-scriptive.

By contrast, in this thesis, we perform a series of empirical studies to characterize the individual constituents of Feedback-Driven Development (FDD): we study the prevalence and evolution of Automatic Static Analysis Tools (ASATs), we explain the "Last Line Effect," a phenomenon at the boundary between ASATs and code review, we observe local testing patterns in the Integrated Development Environment (IDE) of developers, compare them to remote testing on the CI server, and, finally, should these quality assurance techniques have failed, we examine how developers debug faults. We then compile this empirical evidence into a model of how today's software developers work.

Our results show that developers employ the different techniques in FDD to best achieve their current task in the most efficient way, often knowingly taking shortcuts to *get the job done.* While this is efficient in the short term, it also bears risks, namely that prevention and introspection activities fall short: developers might not configure or combine ASATs to their full benefit, they might have wrong perceptions about the amount of time spent on quality-control, quality-related activities such as testing could become an after-thought, and learning about debugging techniques falls short. A relatively rigid, tool-enforced FDD process could help developers in not committing some of these mistakes. Our thesis culminates in the finding that feedback loops are the characterizing criterion of contemporary software development. Our model is flexible enough to accommodate a broad band of modern workflows, despite large variances in how projects use and configure parts of FDD.

Samenvatting

Softwareontwikkelaars van vandaag hunkeren naar feedback over hun werk, danwel van hun peers via code review, via statische analyse tools zoals hun compiler, ofwel via de uitvoering van testen, hetzij lokaal of op afstand in de Continuous Integration (CI) omgeving. De strakke integratie van sociale coding sites zoals GitHub en CI services zoals Travis CI hebben software ontwikkeling enorm veranderd. Met deze grote verschuivingen op het vlak van software ontwikkeling missen we een holistische beschrijving van hedendaagse software ontwikkelingspraktijken. Bestaande beschrijvingen zoals het V-model zijn te grof om een individuele workflow te beschrijven of gaan alleen over een onderdeel van het ontwikkelingsproces, zoals Test-Driven Development (TDD). Bovendien zijn de bestaande modellen meer *pre*- dan *de*-scriptief.

In deze thesis daarentegen doen we een reeks empirische studies om de individuele onderdelen van Feedback-Driven Development te beschrijven: we onderzoeken hoe wijdverspreid het gebruik van Automatic Static Analysis Tools (ASATs) is, bekijken de evolutie van hun gebruik en we leggen het “Last Line Effect” uit, een fenomeen op het snijvlak van ASATs en code reviews. Ook observeren we de lokale testpatronen van ontwikkelaars in hun Integrated Development Environment en vergelijken we die lokale patronen met het op afstand testen op de CI server. Vervolgens bestuderen we hoe ontwikkelaars fouten debuggen in het geval dat de voorgaande maatregelen om de kwaliteit te bewaken falen. Ten slotte verzamelen we het empirische bewijs dat we hebben verkregen om tot een model te komen van hoe softwareontwikkelaars heden ten dage werken.

Onze resultaten tonen dat programmeurs de verschillende technieken in FDD gebruiken om hun programmeeropdracht op de meest efficiënte manier uit te voeren, waarbij ze vaak bewust een shortcut nemen om de klus te klaren. Het valt niet te ontkennen dat die op korte termijn efficiënt is, maar deze manier van werken brengt ook risico’s met zich mee, vooral op het vlak van preventie en introspectie-activiteiten die te kort schieten. Zo kan het voorkomen dat programmeurs hun ASATs niet optimaal configureren of combineren, ze een verkeerde perceptie hebben qua tijdsbesteding van kwaliteitscontrole, ze activiteiten verwant aan kwaliteitsbewaking, zoals testen, als bijkomstigheid beschouwen en zichzelf onvoldoende scholen op het gebied van debuggingtechnieken. Een relatief rigide, door tools gehandhaafd FDD proces kan ontwikkelaars begeleiden om deze fouten niet te maken. Onze thesis culmineert in de vondst dat feedbacklussen het karakteriserende criterium zijn van moderne softwareontwikkeling. Ons model is flexibel genoeg om er een brede waaier aan moderne workflows in onder te brengen, ondanks de grote variatie in hoe projecten delen van FDD gebruiken en configureren.

Acknowledgments

Without a doubt, the acknowledgments are the most widely and most eagerly read part of any thesis. Mine shall not disappoint, either, for this thesis and the time I had while writing it would not have been nearly so good without the contributions, large and small, of many a people.

Contribution-based Acknowledgments

For specific parts of the thesis, I want to acknowledge individuals whom I had fruitful discussions with, who gave me a pointer to a paper I was missing, who reviewed a manuscript, or otherwise provided input that advanced said part or simply me.

Cover: thank you, Zsófia, for being so responsive, fast, and patient with me. You created a stunning piece of art with a strong connection to the thesis. Köszönöm szépen!

Chapter 2: I thank Bastiaan Reijm for the help that he provided throughout the development of UAV, Fabian Beck for useful suggestions on the first release candidate, and all students who participated in our usability evaluation.

Chapter 3: I thank Diomidis Spinellis for an inspiring conversation during ICSE'15 in the "Mercato Centrale." For reviewing drafts of this chapter, I thank Maurício Aniche, Joseph Hejderup, and Mozhan Soltani.

Chapter 4: I thank Mathias Meyer (then-CEO of Travis CI), Arie van Deursen, Felienne Hermans, Alexey Zagalsky, Maurício Aniche, and previous anonymous reviewers for their feedback.

Chapter 5: I owe our biggest gratitude to the hundreds of WatchDog users. Moreover, I thank Maryi Arciniegas-Mendez, Alan Richardson, Nepomuk Seiler, Shane McIntosh, Michaela Greiler, Diana Kupfer, Lars Vogel, Anja Reuter, Marcel Bruch, Ian Bull, Katrin Kehrbusch, Maaike Beliën, and the anonymous reviewers. I thank Andreas Bauer for help with the WatchDog transformer.

Chapter 6: I thank all study participants, who, in spite of showing their fallibility, allowed us to research their debugging behavior. I thank Georgios Gousios and Earl Barr for reviewing this manuscript.

General Acknowledgments

Somewhat impersonally, I want to thank the European Union for paving the road that makes it so easy and enjoyable for foreigners like me to work in a different member state; the Dutch I want to thank for being welcoming, relaxed, and pretty darn awesome (despite the bread), i.e., just being Dutch. I also felt that TU Delft deeply cares about their employees. I had an absolutely delightful four years here. Bedankt allemaal!

Andy: when you offered me to pursue a PhD under your supervision on that sunny October day in 2013, I did not know you well. However, it took little effort to notice that

you seemed to be one of the kindest, most understanding, and open-hearted persons I have had the pleasure to get to know (and I am only using the plural here in case someone else I worked with reads this). I have to say, some four years later, I stand by that sentence with certainty. I did not realize it back then, but the decision to do a PhD with you turned out to be the best possible decision I could have taken. Thank you for giving advice when I needed advice, thank you for being compassionate when I needed companionship, thank you for playing the advocate when I needed a devil, thank you for letting me co-supervise three Master students, thank you for all the sweets, and, above all, thank you for giving me space. Space to fail, space to develop (both software and myself), space to go abroad. In the past years, I never once heard you say “no” to yet another arcane idea from me. For that, I owe you my biggest thank you!

Georgios: I learned about 998 things too many from you to list them all here, so suffice it to say that I am your padawan. If I had to name two things I learned from you, it would be that you showed me that one can never know enough technically and the fine art of sometimes not giving a damn (and especially not to make someone else’s problems your own). I am still learning on both ends, but, hey, I don’t care. The exact place our journey will lead us to, I do not know yet, but it certainly does not stop here. Thanks for being awesome and thanks for being my best friend in Delft!

Arie: thank you for providing an open and friendly environment in which to do cutting-edge research in. In every encounter with you, I perceived you as a fundamentally happy person (in case you noticed: sorry if I sometimes smirked when seeing you. I swear it was because of that!). Thanks for letting me (stay) in your research group and thanks for being so quick to provide constructive feedback to my thesis!

Alberto: writing that first MSR paper on code reviews with you was a transformative experience that I was fortunate enough to have gotten early in my PhD. It showed me how much diligence one should put into composing every single element of a paper and talk, even on secondary material, and that there simply is no good enough (manifesting itself in my habit of submitting incremental improvements of papers well past their deadline).

Tom: thanks for giving me the opportunity to do research at Microsoft and have an absolutely wonderful summer in Seattle. Thank you for allowing me to be very diligent. When you were away, it was very clear that it is you who holds the mini-group at MSR together, always assembles everyone for lunch, and organizes fantastic outings. Thank you for creating an environment in which I could not only work on big data, but also learn from you, all the while having tons of fun! So long and thanks for all the f~~ish~~ood!

Per, Diomidis, Geert-Jan, and Eelco: thank you for accepting to be in my defense committee, bearing with all the Doodles (I’m so sorry!), and traveling to Delft (Per, Diomidis)! I truly appreciate your time and effort.

Annibale: thanks for always having a smile, an open ear, a deep understanding of what is important in life (no question there, it’s food!), and for showing me how to make pasta (I am not writing teaching, because unfortunately, it still keeps sticking to my unworthy, non-Italian hands).

Fenia: thanks for the fantastic food and being a super easy-going neighbor across the “white bridge.” You rock (and sorry for the not-so-useful ancient Greek name suggestions)!

’Ελλη: 🖖

I have had the pleasure to have many joyful conversations with current and past col-

leagues, but some stand out. *Nicolas:* thanks for your hospitality and sense of humor. *Hennie:* thanks, too, for your sense of humor (though totally different than Nicolas's). *Felienne:* thanks for your embracing and enthusiastic character (and all the party invitations ☺). *Bas:* thanks for sharing your running routine with me and being an all-around nice person. *Maria:* thanks for being jolly fun (and always in a good mood!). *Joseph:* thanks for, despite being a super Swede, also being talkative (and a super shopper). *Qianqian:* thanks for being a really pleasant person. *Anja:* thanks for being a great first office mate and the thesis printing info. *Tamara:* thanks for taking care of every organizational detail.

Radjino, Igor, and Niels: thanks for sharing part of the ride and being my master students. It was an absolute pleasure to work and learn with and from you.

Shane, Rolf, Elmar, and Andrey: thanks for co-writing with me. Your contributions were not only important to the respective papers, but more so, I learned skills from you that have shaped how I work today.

Corinna, Evi, Christian, Ernst, und Martin: vielen Dank, dass ihr nach Delft gekommen seid (das bedeutet mir wirklich viel)!

Stefan (Zachseule): thanks for being a good friend. It was great to visit you in Kassel.

Wilma: dankjewel voor het tuinieren en dat je zo een goede buurvrouw bent!

Dino, Peter, Mario: thanks for being awesome ninjas! Peter, you are the genuinely funniest person I have ever met. Keep it up! Dino, thanks for being the best ~~truffle~~ powder pig, for your pleasantly calm personality, and all the (mountain) fun we had and will hopefully be having!

Heiko: your coolness and ability to find joy and relaxation is an absolute inspiration to me. Whenever I am stressed, I should remember I have the chillest friend ever. Wuff!

Petra: bedankt dafür that je mir Dutch geleerd hast. Jij bent echt een belankrijke deel waarom ik het zo ontzettend leuk hier vind (en mij ook een beetje 'thuis voel').

Melanie, Fabi, Benni, Marcel, and Ryan: thanks for being great friends, be it for hanging out, doing sports, cooking, climbing, sending packages, or hiking (Yosemite, Mt. St. Helens!). I hope we will reach many peaks together.

Martin: thanks for being a great host, chef, and mountain guide. My first multi pitch route up the Aggenstein was an absolutely amazing experience with you, duly celebrated.

Thomas (imagine Ali G speaking here): what up, thanks for being my main man! I am truly happy and honored to have you in my life.

Pixie: thanks for hopping (or cycling) aboard and joining the tour (or rather, roller coaster ride)! I am grateful for all the things you showed me and have done for me, and everything we did and will do. ♡

Mama, Papa, und Nora: danke, dass ihr immer für mich da seid, mich immer unterstützt und ich immer auf euch zählen kann. ♡

No PhD is an easy journey. However, thanks to all of you, I can count the days where I did not like what I was doing on one hand, and my memory cell for counting how many times I smiled, thinking "this is exactly what I want to be doing right now," has long since overflown (whether due to memory limitations on my side is left to the judgment of the reader, possibly after assessing the remainder of this thesis). **Thank you!**

Moritz
Delft, January 2018

1 Introduction

In today's software development world, feedback loops pervade the entire life cycle of a piece of code from its inception through its acceptance into the code base to its maintenance life as legacy code. These feedback loops accommodate all stages of quality assurance from human code review to debugging, for at the heart of each loop lies the desire to improve the quality of the examined piece of code by feeding back quality concerns to the developer. We call this highly-flexible process of doing software development on the basis of a configurable number of quality assurance methods Feedback-Driven Development (FDD). In spite of large historical and technical differences between the individual constituents of FDD, have recent advances in collaborative software development enabled the seamless and continuous integration of even such opposed techniques as static and dynamic analyses. As a result, the multitude of feedback loops and the interplay between them has become a characterizing criterion of modern-day software development.

In this thesis, we study the feedback loops that underlie modern software development. We perform empirical research on each of the proposed components of FDD from static analysis tools over code review to testing and debugging via a series of independent case studies. Compiling the findings of these studies under the umbrella of Feedback-Driven Development enables us to build a first reality-grounded understanding of contemporary software development practices in a highly collaborative and integrated development world.

This chapter is partly based on *M. Beller. Toward an Empirical Theory of Feedback-Driven Development, ICSE'18 (Student Research Competition)* [1].

A plethora of breakdowns of software engineers' work processes exist today, ranging from structured, general process decompositions such as the V-model [2], over more flexible guidelines such as the agile manifesto [3] to practically process-free software creation paradigms such as the chaos model [4]. These models, however, tend to focus less on an individual developer's workflow, but more on the general processes to be followed in an entire project. Thus, they are of little help in describing the individual act of creating and improving program code. Other, partly more recent inventions such as Test-Driven Development (TDD) [5] or its off-spring Behavior-Driven Development [6] provide recommendations closer to a single developer, but they often focus on a somewhat limited aspect of the software development process, for example how to drive development via testing, which leaves out other important feedback-cycles such as code review or static analysis. Thus, they cannot provide us with a model capturing a more holistic individual code creation process. A common denominator of all these models is that they are *pre*- rather than *de*-scriptive: they argue that a certain methodology *should be* applied instead of studying what *is being* applied.

In contrast to these pre facto models, we build up our model of Feedback-Driven Development (FDD) post factum based on empirical evidence. We perform empirical analyses on the constituents of today's software development workflow first and then compile this empirical evidence into a model of Feedback-Driven Development. Our model is thus a contemporary mirror of the development practices of software developers.

Gaining this understanding is important because it allows us to adequately reason about current development practices in a precise and defined way. As an emerging hypothesis, FDD shapes our thoughts so that we have a common language to express ourselves eloquently and efficiently about modern development practices. It allows us to educate aspiring students on the state of the art of software creation, to compare the advantages of different implementations of FDD to each other, to identify areas for further research under its umbrella, and to propose further improvements in the current FDD circle.

1.1 Background & Context

In this section, we give an overview of the FDD model and show how it is embedded within the Software Engineering research domain and how it connects to related work.

1.1.1 A Model of Feedback-Driven Development

Today, developers can receive feedback on a piece of code they have created from a variety of sources: the compiler, automated static analysis tools, the Continuous Integration server, local or remote test runs, peers who perform a code review, if necessary, a debugging session that can includes remote logging information or application telemetry. Even end users can give feedback to the developers directly, often via an automated bug monitoring system. The goal of all these different feedback mechanisms is to enable developers to immediately improve the quality of their software.

Figure 1.1 sketches the Feedback-Driven Development workflow alongside these quality assurance methods typically found in today's software development projects and how they relate to other concepts in the Software Engineering domain. Every rounded box represents a concept, possibly grouped together by an overarching theme in a dotted box.

Edges between them represent the typical workflow, while the absence of edges means that there is no fixed order. A dotted edge symbolizes the concept of having an influence on the connected stage. Black stages represent stages covered in this thesis, concepts and relations in gray entities outside the scope of this thesis. We take here the technical perspective of how a code contribution progresses from its initial inception ① to its final rejection or acceptance into the code base ⑤. One short FDD loop for a developer is to go from creating code to testing in the IDE (local) and back to Code Creation to fix a finding caught by the reviewer. However, the model also caters for different workflows, for example to go from Code Creation, over testing locally in the IDE and remotely to accepting the piece of code into the project. This thesis is performs an empirical evaluation of the core of FDD, the Code Quality Assurance methods in Figure 1.1. The act of writing the code itself is outside the scope of this thesis and studied partly in the field of program comprehension [7, 8]. How a code contribution be best packaged, for example as a pull request, and which characteristics a contribution should have for a fast acceptance into the code base [9] also lies outside the scope of this thesis.

There exist various triggers for the creation of a code contribution, be it the need to introduce a new feature, fix a bug, or improve the maintainability of the system [10]. The developers working on a code contribution often obtain more detailed information about its desired nature in explicit and structured form from the system's requirements, often written down as tickets or issues in an Issue Tracking System, more implicitly through discussion with colleagues, other stakeholders, or a (hopefully) mutually shared project vision. Coming up with good requirements and how to translate them into work tasks are questions that concern the research field of Requirements Engineering [11]. Which of these work items to tackle next is the domain of issue prioritization, a sub-field of Software Engineering research that often determines the success or failure of a project [12]. In a development methodology called "DevOps" [13, 14], developers are in charge of running their own code in production. This typically involves monitoring live systems to get feedback of the successful operation of code. However, this feedback loop differs from the inner FDD loop modeled in Figure 1.1 in that it might be a trigger for a code change (or, in fact, a change in operations), but cannot usually be used to decide about the acceptance or rejection of a code contribution ⑤, as this information is only available after the code has been integrated and deployed. Running A-B experiments and (automatically) deciding on their outcome in production, like Google and Bing do to test the efficiency of certain changes [15, 16], of course somewhat softens this clear separation.

Apart from the quality assurance methods in Figure 1.1, developers can also receive feedback from other "soft channels" such as fora by asking for help or ideas from their peers. While this bears some resemblance with Code Review, we modeled it as a separate entity because in contrast to Code Review, it is not mandatory to use these channels or fora, developers might or might not submit (sample) code with their original question, and answers are of more ad-hoc nature. In spite of the inherent randomness and seemingly unpredictable nature of the process [17], practitioners have referred to the prime example of such a feedback source, Stack Overflow, as "game changing" and "the biggest invention in Software Engineering in the past decade" [18]. We have refrained from studying Stack Overflow as part of this thesis in light of its different nature in comparison to the other FDD stages and an abundance of empirical research on it [17, 19–21]. Outside of

1

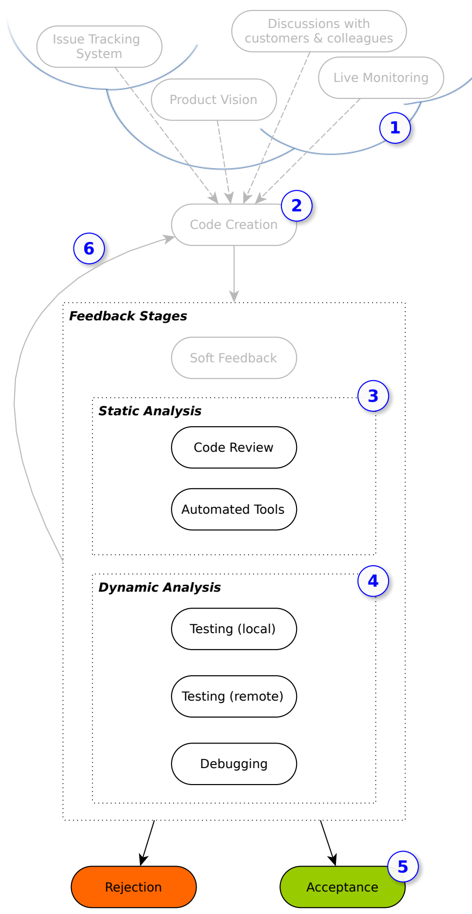


Figure 1.1: The stages of the FDD model and their relationship to other Software Engineering concepts.

the scope of this thesis is also the study of the "Code Creation" stage ②. The Incremental Change process [22] complements FDD by describing what happens there.

We divide quality assurance methods of FDD, which we study in this thesis, into two complementary groups that work fundamentally differently on a technical level:

1. Static Analysis ③ examines program artifacts or their source code without executing them [23], while

2. Dynamic Analysis ④ relies on information gathered from their execution [24].

Static Analysis not only includes so-called Automated Static Analysis Tools (ASATs), which perform property checks on the software without human interference, but also includes manual assessment in the form of code review [25]. In particular Modern Code

Review is a topic of active research in the Software Engineering community [26, 27]. Dynamic Analysis, on the other hand, is not confined to testing the software, but also includes debugging, which routinely involves reading run-time log messages or analyzing real-time dashboards in the case of remote systems. It is customary for contributions in the making to go through a cyclical review process until they reach a pre-defined acceptance criterion ⑤. Consequently, most projects explicitly allow reworking and perfecting contributions after their initial submission ⑥. These feedback loops thus stand at the heart of modern software development. The precise order of quality assurance checks in Figure 1.1 may deviate from project to project and even feedback cycle to feedback cycle. For example, in an attempt to minimize human involvement, many projects do not perform mandatory human code reviews [27] or defer them until remote testing on the Continuous Integration server has shown that the contribution has reached a certain degree of quality. Because the output of ASATs or even compilers can be hard for developers to interpret [28], some projects such as Ruby on Rails have set up advanced bots that reply in a style that makes them almost indistinguishable from a human reviewer [29], shown in Figure 1.3. Examples of such bots that bridge the gap between the way a human and an ASAT reports their findings are Microsoft's review bot and lgtm.com, which provides "automated code reviews for developers [with] [d]eeper insights [and] [a]ctionable results."

1.1.2 The Case for FDD in a Collaborative Coding World

In today's ever more collaborative software development world [30–32], most notably currently seen in the GitHub ecosystem, the number and quality of code contributions from developers outside the core development team poses a particular challenge to projects [33, 34]. Even companies such as Microsoft who traditionally used to be skeptical of OSS [35] have recently started to embrace the Open-Source Software model [36, 37], largely increasing the visibility of their projects and the number of potential contributors to them. Simultaneously, project maintainers, who are in many cases volunteers and do this important service to the world-wide software community in their free time, have started to suffer from an increasing workload caused by an overwhelming number of pull requests.

Many code contributions in today's Open-Source world come from one-time contributors [38]. These pull requests from project outsiders are potentially of low quality or not aligned with the project's direction [33, 39, 40]. Ensuring a good fit with the project is particularly challenging and important for these contributions. Data extracted from GHTorrent [41] shows the widening gap between the number of opened and merged pull requests in Figure 1.2. By September 2017, a total of 1,653,879 pull requests on GitHub were open, but neither merged nor closed by the project maintainers. When we transfer this situation to our model, many of these potential contributions would be stuck in one of the stages ② to ④ of Figure 1.1.

Drowning in pull requests or issues is not a problem that evenly spreads across the more than 20 million GitHub projects, but targets precisely the important and well-known projects: As one such example, the Ruby on Rails project had 719 open pull requests on November 29, 2017. The trend does not affect the large amount of dormant toy, private or forked projects on GitHub, since with small interest in a project come few pull requests. On a project level, it means that it takes an unnecessarily long time for code contributions

to finally make it to the project's code base, at which point extra effort might be necessary to re-base the contribution. This is a frustrating situation for contributors, maintainers, and users that slows down collaboration and innovation. It shows the fundamental dilemma that undermines the OSS community: Most projects which would need outside contributions never receive any, and few projects are flooded by them.

Thus, automating the feedback stages in FDD could help both active projects by reducing their workload and the many dormant projects to which developers cannot ensure the quality of their code contribution because the maintainer is not available anymore.

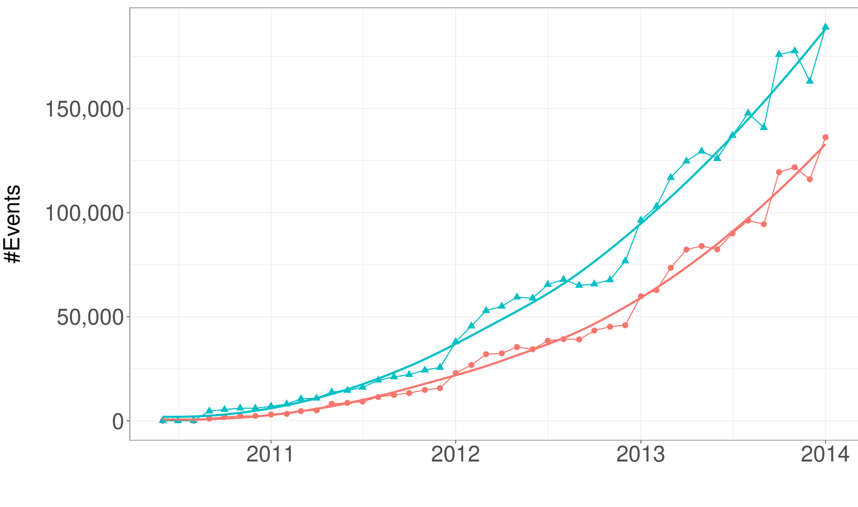


Figure 1.2: Mind the gap: Number of opened and merged pull requests on GitHub from 2010–2014.

1.2 Feedback-Driven Development in Practice

Figure 1.3 shows FDD's feedback loops at work on an exemplary constructed pull request in the Ruby on Rails project. The author `witlessbird` might have drawn inspiration for the creation of the pull request "Initial support for running Rails on FIPS-certified systems" Ⓐ from issue `#31203`, which stated that "Rails is not compliant with FIPS 140-2 mode" (step ① in Figure 1.1). Within seconds of the first code change Ⓑ, the above discussed Rails bot hails the newcomer and assigns a suitable reviewer Ⓒ. The bot's comment also contains a warning about the automated use of Code Climate, an ASAT that let the integration of the initial commit fail Ⓑ. With these hints, the contributor goes back to code creation ② and authors another commit that passes all checks Ⓓ. Following the ASAT-feedback cycle, a code review round begins with the suggestion of replacing a hashing algorithm Ⓔ, the implementation of which promptly follows. Another reviewer jumps in and asks for more changes in a constructor Ⓕ.

However, during the course of implementing changes suggested in the code review feedback loop, the build broke Ⓖ. The author now enters the inner-remote testing loop ④

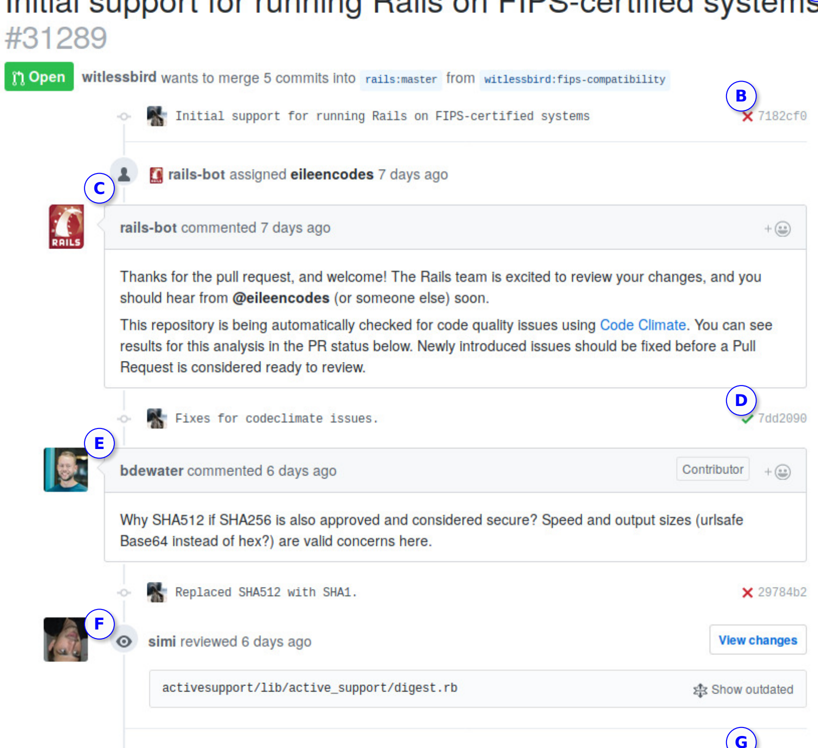


Figure 1.3: A constructed examplary pull request of the Ruby on Rails project showing feedback loops and their integration into pull-based development in action.

in Figure 1.1. A first try at fixing the build fails Ⓗ. Upon receiving this feedback, the second try succeeds. The contribution is awaiting final acceptance ⑤. The complete sequentialization of this process via Figure 1.1 is thus: Issue Tracking System ⟶ Code Creation (by `witlessbird`) ⟶ Automated Tools (review bot) ⟶ Automated Tools (Code Climate, failed) ⟶ Code Creation (by `witlessbird`, fix Code Climate warnings) ⟶ Code Review (by `bdewater`, first reviewer) ⟶ Code Creation (by `witlessbird`) ⟶ Code Review (by `simi`, second reviewer) ⟶ Code Creation (by `witlessbird`) ⟶ Testing (remote, failed, Travis CI) ⟶ Code Creation (by `witlessbird`) ⟶ Testing (remote, failed, Travis CI) ⟶ Code Creation (by `witlessbird`) ⟶ Testing (remote, success, Travis CI). However, this example also shows us the limitation of a repository analysis alone: we cannot determine from it whether the developer `witlessbird` entered the local testing or debugging loops, for which we would need telemetry data from their computer.

At the end of this workflow stands a decision on whether or not the code contribution makes it into the code base of the project ⑤. The advent of distributed version control systems such as BitKeeper, Mercurial, or git has allowed projects to work in novel collaborative ways. Many projects have adopted a pull-based development model [9], both in the Open-Source and the Closed-Sourced world. Pull-based development means that contributors make their own copy of a project's repository they want to contribute code to, a so-called fork, perform changes on their fork, and finally file a pull request asking that the changes from their copy be merged back into the main repository. "Then, the members of the project's core team (the integrators) are responsible for evaluating the quality of the contributions, proposing corrections, engaging in discussion with the contributors, and eventually merging or rejecting the changes." [33] Not only are pull requests thus open calls for code review, but they also allow the structured, automated, and efficient integration of both static and dynamic checks of the contribution on platforms such as GitHub. A merged pull request equates an accepted code change in Figure 1.1.

1.3 Research Goal and Questions

In addition to an always existing desire to automate labor-intense and error-prone human work in Software Engineering, the trend to more outside contributions, overwhelmed project maintainers, and a desire for more reliable software systems necessitates automating and improving projects' feedback loops. Before we can suggest meaningful improvements to this process, however, we must first develop a thorough understanding of it. Moreover, knowledge itself can be a way to improvement, as we show on several occurrences in the remainder of this thesis.

> This thesis is concerned with the empirical assessment of the state of the art of how developers drive software development with the help of feedback loops.

RQ 1 How do developers use static analysis within FDD?
Step ③ in Figure 1.1, Chapters 2 and 3

RQ 2 How do developers use dynamic analysis within FDD?
Step ④ in Figure 1.1, Chapters 4 to 6

By answering the research questions, the thesis culminates in the first definition and empirical characterization of what we call the "Feedback-Driven Development" process. In it, we compile our findings on the different aspects of the various quality assurance methods into a coherent initial model.

1.4 Research Methodology

In this section, we describe the main research methods we use throughout this thesis and their ethical implications.

The methodological foundations of this thesis are rooted in Empirical Software Engineering, a relatively young sub-discipline of Computer Science that can trace its beginnings to the 1970s [42]. Fundamentally, Empirical Software Engineering applies the sci-

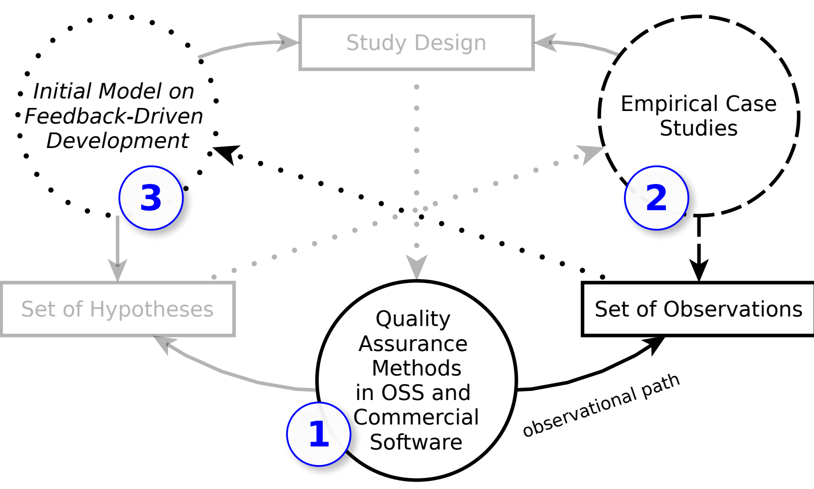


Figure 1.4: Instantiation of the RPS with our research. We followed the observational path [49].

entific method known from the natural sciences (most notably experimental physics) to gain falsifiable insights into various aspects of the Software Engineering domain. At the heart of empiricism applied to Software Engineering lies the idea that to understand the impact of proposed changes, be they human-, tool-, or process-oriented, one must first assess and understand the status quo. Surprisingly, while Software Engineering used to have no shortage of suggestions for arbitrary and sometimes questionable improvements, combined with dubious research practices [43], it lacked empirical evidence in some of its key areas [44–47], a theme that recurs throughout this thesis.

1.4.1 Research Method Categorization

McGrath divides research methodologies into four different quadrants with a canon of eight associated concrete research methods [48]. Which of the eight methods is best suited for a given research objective depends on the desired degree of generalizability, precision, and realism for that study. These range from laboratory experiments, which maximize precision, to formal theories, which maximize generalizability. For the studies in this thesis, we selected mostly research methods in the "Field Strategies" quadrant, thus maximizing realism, arguably the most critical concern for an applied discipline such as Software Engineering. Our study methods comprise field and case studies, but we borrow concepts from sample surveying, computer simulation, and formal theories for improving the generalizability of our findings.

In contrast to McGrath's general research methodology descriptions, the Research Path Schema (RPS) is an analytical framework tailored to Software Engineering. It allows Software Engineering researchers to clearly communicate the principal setup of their research to their peers. It also describes a way to theory building via different research paths [49]. Depending on the chosen research path, the three domains ins RPS – the sub-

stantive, conceptual, and methodological domain – become the primary, secondary, or tertiary concern of a study. Our instantiation of the RPS in Figure 1.4 starts from the substantive domain "Quality Assurance Methods in OSS and Commercial Software" ①, makes observations by means of a large-scale case study ② and derives a set of hypotheses on Feedback-Driven Development that together form an initial theory ③.

1.4.2 Enablement of Large-Scale Studies

To further increase their generalizability, we perform our studies in a large-scale fashion, typically on hundreds of projects or developers. This brings with it a number of complexities, from recruiting study participants over gathering large amounts of data to processing it, a sub-field of Computer Science touted "Big Data" [50]. A point of criticism against large-scale analyses is that findings would sacrifice deep for broad understanding. However, more involved analyses can also provide deep insights when tailored to specific projects, for example in the form of individual project reports. Moreover, single-project or small-scale analyses cannot uncover general Software Engineering phenomena and thus fail to quantify how widespread a certain issue or how large its impact is. Large-scale analyses help us single out individual problems from issues that plague Software Engineering as a craft, and thus sharpen which problems Software Engineering researchers should tackle.

The scale and nature of our studies (② in Figure 1.4) almost forbids manual observation. Instead, they require a heavily tool-supported approach. We followed this in two ways:

1. We relied on a Mining Software Repositories approach, a sub field of Software Engineering that extracts knowledge from analyzing historic information structured in traditional software repositories such as git or in new data sources such as TravisTorrent. We used the high accuracy of the information embedded in the repositories to improve the precision of our studies.

2. Not all information about feedback loops is present in readily available repositories. While code as the artifact of developers' work is available, it does not give us information about the fine-grained path of how exactly they created that code ② in Figure 1.1. To learn about developers' testing and debugging behavior, we automatically collected their testing- and debugging-related actions by instrumenting their IDEs with telemetry plugins.

Both techniques are scalable, robust, and updateable, causing minimal interference with the usual work habits of developers, thus increasing realism. Effects of a (physical) onlooker or researcher taking notes have been studied extensively in psychology and medicine. Examples are the Hawthorne [51] and trial [52] effects, which describe the phenomenon that participants tend to behave differently when under examination, typically by outperforming their normal baseline in experiments. We reduce these biases as much as possible by using low-interference telemetry methods that do not require the physical presence of a researcher and allow participants to stay in their normal, basically unchanged virtual environments.

To analyze the data we gathered, we employ methods from the fields of data visualization, descriptive statistics, statistical hypothesis testing, and probability theory [53]. We

Table 1.1: Research methods used for each study.

Study	Chpt.	Quant. Analysis	Survey	Interviews
State of ASATs	2	✓		✓
Last Line Effect Explained	3	✓		✓
WatchDog IDE Testing	4	✓	✓	
Travis CI Remote Testing	5	✓		
WatchDog IDE Debugging	6	✓	✓	✓

enrich these methods with explanatory methods borrowed and adapted from the social sciences and known under the umbrella of Grounded Theory. Grounded Theory is "a general methodology with systematic guidelines for gathering and analyzing data to generate middle-range theory" [54]. From the wide range of methods available in Grounded Theory, we use surveys, interviews, and card sorting. In line with McGrath [48], we employ a series of mixed-methods studies that combine several of the above techniques to answer one research question. Particularly for validating the accuracy of survey answers, we triangulate answers in questionnaires with the according data extracted from IDE telemetry. Table 1.1 gives an overview over which research methods we employed for each of the individual studies.

1.4.3 Ethical Implications

Performing research inherently has ethical implications. Shamoo and Resnik describe a responsible conduct of research along several dimensions such as honesty, objectivity, carefulness, openness to share data and results, legality, and human subjects protection, which we strove to adhere to during this thesis [55].

We can principally divide our study methods into ones which operate on openly available data and ones for which we actively collected new data. While both can be sensitive – think of the discovery of a hidden implication revealed by our analysis of freely available data – the analysis of repository data has been standard practice in the Software Engineering community for many years. Typically, the focus of such data is technical (for example, the Travis CI build logs in Chapter 5) and of such low sensitivity to individuals that it poses no risk to them. Moreover, in our studies on openly available data in Chapters 2 and 5 we usually abstract away from individuals to a group of developers working on a project or anonymize them (Chapter 3). Similarly, surveys or interviews on technical subjects typically pose minimal risks to participants as long as they are free to quit at any time and are therefore often exempt from explicit ethics approval.

The collection of fine-grained developer interactions with WatchDog in Chapters 4 and 6 requires more thought. With WatchDog, we followed four principles:

- **Informed Consent**
  All participants actively have to seek and install WatchDog and agree (at least two times) to our privacy policy, once when installing the plugin in the IDE, and once when registering a user. Moreover, the website also contains a detailed explanation of how and which data WatchDog gathers and for which research purposes we

plan to use it. Participants can stop using WatchDog or remove their data at any time.

- **Voluntary Registration**
  Developers could use WatchDog anonymously, without having to fill-in a registration form.

- **Least Amount of Data Gathering**
  With WatchDog, we followed two aims: Explore developer testing and debugging. While it would have been technically easy to log all user interactions with their IDE, we explicitly constrained WatchDog to only the data for which we had hypotheses and which was the focus of the ongoing research projects. This also meant sacrificing possible future research opportunities.

- **Early Anonymization**
  To protect the intellectual properties of individuals and companies, we designed WatchDog to never transfer any actual content, that is neither file content nor file names. To differentiate files and projects, we hash file and project names, a one-time, irreversible operation. The design of WatchDog ensures this information never leaves the participants' computers, since the hashing happens at the client side and the connection to the server is secured.

Finally, the Human Research Ethics Committee of TU Delft granted retrospective approval of WatchDog on May 8th, 2018, under application number 416 "TestRoots Watchdog (Updated)." It categorized WatchDog as "minimal risk."

1.5 Replicability, Open Science & Source

Open science is the "movement to make scientific research, data and dissemination accessible to all levels of an inquiring society." [56] In the Netherlands, the Dutch Funding Agency NWO demands that every research result funded by the public also be accessible by the public [57]. In this section, we describe how the contributions in this thesis support this aim.

1.5.1 Open Data Sets

Not only are all publications embedded in this thesis under open access, but so are the data and source code contributions associated with them, to the extent data licensing and privacy agreements allow. Our goal is to foster replicability and invite other researchers to build on our work. To this end, we have performed all of our studies at least partly on freely available OSS projects. Table 1.2 shows which studies include OSS and which include additional closed-source projects. OSS of course does not equate "non-professional" or "non-commercial," as in many cases of successful OSS projects, a business or a professionally organized body such as the Apache or Linux foundations stand behind them. Closed-source projects in our studies comprise a variety of proprietary sources, from commercial systems to private personal projects.

With a trend toward freely accessible data comes additional responsibility on the authors to also make this data consumable for others. The good practice of this is often

Table 1.2: Overview of data sets, type of included projects, and replication packages.

Data set	Chpt.	Host	OSS	Non-OSS	Size	CC-License
ASAT Mapping / GDC	2	`FigShare`	✓		1 MB	BY 4.0
Micro-Clones	3	`FigShare`	✓		57 MB	BY 4.0
WatchDog Test	4	-	✓	✓	34 GB	-
TravisTorrent	5	`Website`, `Archive.org`, `BigQuery`	✓		3.2 GB	BY-NC-ND 3.0
Travis CI Build Logs	5	`Website`	✓		1.5 TB	-
WatchDog Debugging	6	`Archive.org`	✓	✓	388 MB	BY-NC-ND 3.0

referred to as "data stewardship" and summarized in the four FAIR principles [58], which we have followed in the creation of our data sets listed in Table 1.2.

Findable: We host our data sets on search-engine indexed services such as FigShare or Google BigQuery and gave them human-readable names like TravisTorrent.

Accessible: Long-term storage solutions, The Internet Archive, FigShare, or Google BigQuery host versioned archives of our data.

Interoperable: Our data sets use exclusively open file formats such as `csv` and can thus easily be combined with other data.

Reusable: Our data sets are either self-descriptive or come with extensive descriptions in standardized formats (for example, ISO dates). In addition, we licensed all data sets over which we have authority under Creative Common licenses [59].

1.5.2 Open-Source Contributions

This thesis has also lead to the creation of a number of open-source contributions, primarily the open-sourcing of infrastructure and analysis code to drive the research projects, but also secondary patches to third-party tools. Table 1.3 gives an overview of these contributions.

During the course of this thesis, we have created and maintained the active Open-Source projects TestRoots WatchDog and TravisTorrent and contributed a number of deployed patches to Open-Source software to address issues we found during our research (for example, in the parsedate,[1] cloc,[2] or Samba[3] projects) or to add new features to accommodate our research, as in the case of creating a dedicated marketplace for experimental scientific extensions (Eclipse).[4,5] Last but not least, projects such as Eclipse[6] or

[1] `https://github.com/gaborcsardi/parsedate`
[2] `https://github.com/AlDanial/cloc/issues/153`
[3] `https://bugzilla.samba.org/show_bug.cgi?id=12373`
[4] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=450853`
[5] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=451221`
[6] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=498469`

Table 1.3: Overview of main code contributions (measured with CLOC).

Project	Chpt.	Host	#Commits	SLOC	Main Language
ASAT Config.-Anal.	2	GitHub	69	1,504	Java
ASAT-History-Anal.	2	GitHub	9	344	C#
UAV	2	Website, GitHub	1,127	32,355	Java
Last Line Analysis	3	FigShare	10	307	R
WatchDog	4, 6	Website, GitHub	1,042	13,859	Java
WatchDog Pipeline	4, 6	GitHub	1,015	10,489	R
TravisTorrent	5	Website, GitHub	316	8,627	Ruby
Σ			3,588	67,485	Java, R, Ruby

Facebook's ASAT PFFF[7] started to implement features or improvements that arose based on results of our research.

1.6 Outline & Contribution

In this section, we first outline the structure of the thesis by summarizing each chapter and referencing its originating publications. We then describe contributions performed during the course of this dissertation that are not included in this thesis. Figure 1.5 provides a graphical overview of the thesis contents. The most frequently occuring terms in this thesis compose its word cloud.

In this thesis, we define and describe different constituents of the Feedback-Driven software development process depicted in Figure 1.1. This model of the Feedback-Driven Development process guides the reader through the remainder of thesis. Figure 1.6 enriches the code quality assurance methods in Feedback-Driven Development with the papers that study the associated topic. We start our investigation of FDD with an overview analysis of how automated static analysis tools have been picked-up by state-of-the-art GitHub projects. Given that some ASATs such as FindBugs and Checkstyle have existed for over 10 years, it seems high time to assess their practical influence. In many FDD implementations, after an initial automated assessment follows a manual code review. The Last Line Effect is a phenomenon we first became aware of through such manual code review. While the effect itself is an observation outside the FDD model, we were able to automatize its detection via an ASAT, making it possible to detect it consistently as part of the "Automated Tools" stage in FDD. This also demonstrates the growing importance of automating feedback via ASATs in FDD. Having studied static analysis, we then turn to the dynamic analyses in FDD. We begin by studying developer testing as close as possible to its origin, namely the developers' IDE. Having gained a picture of the intermittent nature of developer testing locally, we study how the naturally more structured remote quality assurance on the CI server compares to it. We call it more structured, since – if

[7] https://github.com/facebook/pfff/pull/143/files

Figure 1.5: Cover variant featuring a word cloud based on the contents of the thesis (using wordclouds.com).

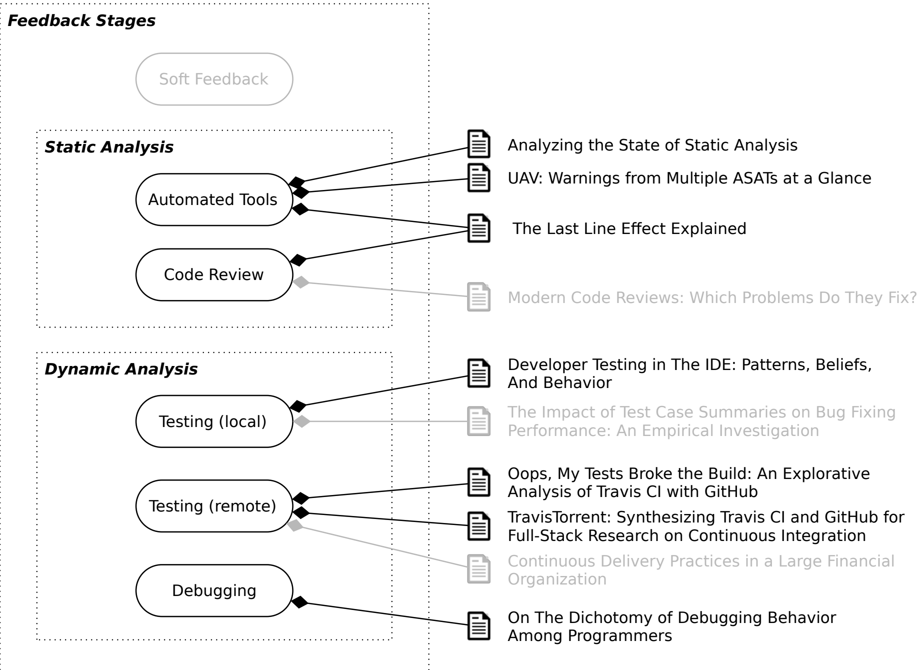


Figure 1.6: Inner stages of the Feedback-Driven Development model from Figure 1.1, annotated with their associated publications. We (co-)authored the grayed-out papers during the course of creating this thesis, but chose not to include them because we were not the leading author or because the research was done as part of our Master's thesis.

configured properly – it runs tests and possibly ASATs automatically with every commit. Finding the answer to whether tests can effectively reduce the debugging burden is part of our investigation of the last dynamic analysis stage in FDD, debugging. In many cases, debugging only happens when the previous stages have somehow failed. This special position in FDD makes it an interesting field of study to complete our empirical study of FDD.

1.6.1 Thesis Structure

This thesis is portfolio-based, comprising a series of independently published articles. We have adapted these articles and in some cases merged them together to build a cohesive thesis, but kept their principal organization intact to allow for an easy mapping of the chapters to their originating papers. In Figure 1.6, we associate each stage in FDD with the scientific articles that cover the topic of this stage. The order of topics in the figure mirrors the order of chapters in this thesis. All articles are freely available under green open access from TU Delft's repository `pure.tudelft.nl` and linked to in their associated bibliography entries.

- Chapter 2 studies how projects on GitHub use ASATs. We evaluate their prevalence

in state-of-the-art projects on GitHub. By performing a history analysis on their configuration files, we can identify how much developers customize and adapt them throughout the evolution of the project. This chapter also introduces the tool UAV, which we created to help developers and researchers with some of the issues uncovered in our empirical analyses. The chapter's body is based on our SANER'16 paper "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software" [60] and incorporates our tool paper on UAV "UAV: Warnings from Multiple Automated Static Analysis Tools at a Glance" [61], which won the SANER'17 best tool demonstration award.

- In Chapter 3, we define the "Last Line Effect," the startling realization that the last line or statement in a micro-clone is more likely to contain an error than all previous lines taken together. This phenomenon lies on the intersection between manual code review and ASATs: Checks conceived during our research and implemented in PVS-Studio and Facebook's pfff now allow developers to find instances of this type of fault automatically. However, we first became aware of these types of faults through manual code review. Collaborating with a psychologist allowed us deeper insight into possible reasons for the existence of the last line effect, for which we found no apparent technical explanations. We published this chapter as the invited EMSE journal extension "The Last Line Effect Explained" [62] in 2016, based on our ICPC Early Research Achievements (ERA) paper "The Last Line Effect" [63], which won the best short paper award in 2015.

- Chapter 4 presents a study on how developers use the immediate testing feedback loop in the IDE to guide the development of their software. We call this subordinate loop of the FDD cycle "Test-Guided Development." The study is based on telemetry information from more than 2,400 participating developers that we measured from within four IDEs with our WatchDog and FeedBaG++ plugins over the course of more than 2.5 years. The chapter content comes from our TSE'17 article "Developer Testing in the IDE: Patterns, Beliefs, and Behavior," [64] which is an extension of an ICSE New Ideas and Emerging Results (NIER) paper [65], an ESEC/FSE'15 technical research paper [66], and an SER&IP workshop paper [67]. The NIER paper pitched and demonstrated the feasibility of the original WatchDog idea by empirically studying how a relatively small study population of Computer Science students tested in their IDEs. The SER&IP paper shows by example of the WatchDog plugin family how academic Software Engineering researchers can efficiently run generalization studies despite the limited time and development resources typically available to them. It proposes an approach that relies on reducing maintenance effort and increasing commonality between the different plugin instantiations.

- In Chapter 5, we focus on remote testing on the Continuous Integration server. We compare projects written in a statically-typed and a dynamically-typed programming language. This study also triggered the creation of TravisTorrent, which provides free and open buildlog analytics. This chapter comprises the MSR'17 technical research paper "Oops, My Tests Broke The Build: An Explorative Analysis of Travis CI with GitHub" [68] and the proposal "TravisTorrent: Synthesizing Travis

CI and GitHub for Full-Stack Research on Continuous Integration" [69], which won the call for bids for the MSR Mining Challenge 2017.

- Chapter 6 regards another dynamic feedback loop, namely debugging: developers pose questions about a certain program behavior that they wish to answer by debugging their program. In this chapter's mixed-methods study, we triangulate data from a debugging survey and WatchDog 2.0 telemetry from developers' IDEs with concluding interviews of debugging experts. This chapter is to appear as the technical research paper "On the Dichotomy of Debugging Behavior Among Programmers" at ICSE'18 [70].

In all the above publications, we (the author of this thesis) are the first and lead author, with the exception of "UAV: Warnings from Multiple Automated Static Tools at a Glance," in which we guided a group of Bachelor students to their first publication.

1.6.2 Other Contributions

Apart from the publications included as part of this thesis, we co-authored a number of papers that we shortly describe in the following.

- Our MSR'14 paper "Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?" [27] contains an empirical study into which types of problems developers actually fix during code review.
- In the ICSE'16 technical research paper "The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation" [71], we present an automated approach to generate natural text test case descriptions of automatically generated test cases. In a controlled experiment, we could show that the presence of these descriptions improves the ability of participants to find bugs with the help of the otherwise identical test cases.
- The ICSME'16 industry track paper "Continuous Delivery Practices in a Large Financial Organization" [14] addresses the use and adoption of CI at ING Netherlands by means of a survey among 152 developers.
- Our ICSE'17 introspection-track paper "Double-Blind Review In Software Engineering Venues: The Community's Perspective" [72] reports on how we as researchers could improve the peer review process to make it more objective and less susceptible to apparent and hidden biases. To this end, we surveyed the Software Engineering community's perception of costs and benefits for the introduction of a review process in which reviewers do not know a paper's authors. While no major Software Engineering conference employed a double-blind review process when we launched our investigation in 2015, by 2017, the two major and many second- and third-tier conferences in Software Engineering had switched to a double-blind review model.

Based on research presented in this thesis, we also made a number of non-academic contributions. We disseminated our findings outside the scientific community in popular-scientific developer articles: We authored two articles in the German print magazine “Eclipse Magazin” [73, 74] (Figure 1.7), wrote two posts in Travis CI’s official blog [75, 76] and one in the IEEE Software Blog [77] as a guest author.

Figure 1.7: Eclipse Magazin 1/2015.

2

Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software

The use of automatic static analysis has been a software engineering best practice for decades. However, we still do not know a lot about its use in real-world software projects: How prevalent is the use of Automated Static Analysis Tools (ASATs) such as FindBugs and JSHint? How do developers use these tools, and how does their use evolve over time? We research these questions in two studies on nine different ASATs for Java, JavaScript, Ruby, and Python with a population of 122 and 168,214 open-source projects. To compare warnings across the ASATs, we introduce the General Defect Classification (GDC) and provide a grounded-theory-derived mapping of 1,825 ASAT-specific warnings to 16 top-level GDC classes. Our results show that ASAT use is widespread, but not ubiquitous, and that projects typically do not enforce a strict policy on ASAT use. Most ASAT configurations deviate slightly from the default, but hardly any introduce new custom analyses. Finally, most ASAT configurations, once introduced, never change. Changes are small and have a tendency to occur within one day of the configuration's initial introduction. Our custom-built Unified ASAT Visualizer creates an intuitive visualization that enables developers, researchers, and tool creators to compare the complementary strengths and overlaps of different Java ASATs.

This chapter is based on

- *M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software, SANER'16* [60] and
- *T. Buckers, C. Cao, M. Doesburg, B. Gong, S. Wang, M. Beller, and A. Zaidman. UAV: Warnings from Multiple Automated Static Analysis Tools at a Glance, SANER'17 (Tools)* [61].

Automated Static Analysis Tools (ASATs) scan the source or binary code of a software system for a set of pre-defined problems. ASATs can be configured to detect: (1) *functional* problems, such as resource leakage or incorrect logic; and (2) *maintainability* problems, such as non-compliance with best practices or violations of style conventions.

Next to testing and manual code review, ASATs have become an important pillar of modern Software Quality Assurance approaches. By heeding the warnings that are reported from ASATs, development teams can address problems before they escape into released versions of their software. Coding standards such as NASA's JPL C [78] and Java [79] standards mandate the use of ASATs during the development process; stronger still, they require that crucial software components be free of any ASAT warning.

However, aside from anecdotal evidence, little is currently known about whether and how rigorously developers use ASATs in the ecosystem of Open-Source Software (OSS). A deeper understanding of the real world application of ASATs can guide researchers' future work, help ASAT developers adapt their offerings to better fit their user base, and ultimately improve the user experience of ASATs.

In this chapter, we set out to understand the prevalence of ASATs, their configuration in real software systems, and how those configurations evolve. To study the prevalence of ASATs, we quantitatively and qualitatively analyze 122 popular OSS projects from the GitHub, OpenHub, SourceForge, and Gitorious software forges in search of the use of popular ASATs. In a second study on the configuration and evolution of ASATs, we: (1) produce a *General Defect Classification* (GDC) to map the specific problems that are detected by the 9 studied ASATs derived from the first study to a common format; and (2) analyze how 168,214 OSS projects configure the studied ASATs with respect to the GDC. We address the following broad research questions:

RQ I.1 *How common is the use of ASATs in practice?*
Half of the state-of-the-art OSS projects already employ automated static analysis, although they typically use only one ASAT in an ad-hoc fashion, where the ASAT is not integrated with the flow of development.

RQ I.2 *How are ASATs configured?*
The ASAT configurations in the studied OSS projects barely deviate from the default ASAT configuration and rarely introduce custom checks.

RQ I.3 *How does the use of ASATs evolve?*
Most ASAT configurations, once committed, never change. The ASAT configurations that do change are typically only very slightly modified within the first week of their appearance in the studied repositories.

The remainder of this chapter is structured as follows. Section 2.1 situates this study with respect to the literature on ASATs. Section 2.2 provides the rationale for our research questions. Section 2.3 presents the results of our prevalence study (RQ I.1). Section 2.4 provides an overview of our GDC, while Section 2.5 leverages this classification to analyze ASAT configuration (RQ I.2) and evolution (RQ I.3). Section 2.6 discusses the broader implications of our results. Section 2.7 presents UAV, a Java tool we built based on our findings, and, finally, Section 2.8 draws conclusions.

2.1 Related Work

We first review existing research on ASATs and discuss how it is related to our study of the prevalence and the use of ASATs. Finally, we give an overview of the classifications that the GDC builds upon.

2.1.1 Automatic Static Analysis Tools

ASATs traditionally use techniques such as data-flow analysis and control-flow analysis to find defects in source code [80–82]. However, because these techniques do not scale at large, abstractions have to be introduced [81]. These abstractions, plus the fact that checking common properties of programs is an undecidable problem [83], lead to *false positives*, warnings about defects that do not exist, and *false negatives*, when warnings about actual defects are missing.

While false negatives impact the efficiency of ASATs because defects are missed, false positives cause developers to waste time investigating incorrect warnings in the code. Deciding whether a warning is a real defect or a false positive takes three to eight minutes on average [84–86]. As there can be as many as 50 false positives for every accurate warning [87], analyzing warnings is a time-consuming activity. In their Faultbench data set for research on prioritizing and classifying ASAT warnings across a number of different projects and domains, Heckman and Williams observed roughly 40 warnings for every thousand lines of code [88]. This overload of warnings is a prime reason for developers to avoid using ASATs [89]. While researchers have studied the reasons why developers do or do not use ASATs, there is little data on the prevalence of ASATs in practice. In this study, we therefore quantitatively investigate the state of adoption of ASATs in OSS projects.

Many ASATs differ in the type of defects that they detect. However, even when tools focus on uncovering the same defect type, the variance in defects found is still very large [83, 90–92]. These results indicate that using several ASATs has benefits over using just a single ASAT. However, this increases the number of warnings that developers need to investigate. Thus, deciding to use multiple ASATs is striking a balance between an improved defect detection rate and the additional investigation effort of an increased number of warnings. We aim to determine how common the use of multiple ASATs is.

To better deal with a large number of warnings, studies have investigated ways to prioritize them [86, 88, 93–95]. This has the advantage that a developer can decide how many warnings to analyze based on the importance of the warnings. In lieu of those ranking algorithms, developers can use configuration files to indicate which rules they consider important. This can reduce the number of warnings generated and suppress rules that are prone to emitting false positives. Another reason to study developer preferences is to observe if the use of ASATs reflects their potential. Wedyan et al. and others observed that 15% of all detected defects were functional and the rest maintainability-related [92, 96, 97]. Many studies observed that ASATs rarely find any functional defects [92, 97–100]. In this paper, we analyze a large number of ASAT configuration files to see how these previous observations are reflected in them.

2.1.2 Defect Classifications

In 1993, the IEEE published a standard for classifying defects [101]. It served as the basis for IBM's Orthogonal Defect Classification (ODC) scheme [102]. This scheme uses the de-

fect type as one of the aspects from which to classify the defect. While the ODC scheme has been used in research [103, 104], several studies conclude that it was too abstract and required adaptations to fit any particular use in practice [27, 91, 105]. In this paper, we propose the General Defect Classification (GDC), a remote ODC-descendant that is a generalization of the scheme refined by Beller et al. [27] and Mäntylä and Lassenius [105]. Its ancestry can be traced back to the work of El Emam et al. [106]. Central to this genealogy of classifications is their high inter-rater reliability. The GDC, in contrast to its predecessors, is specifically tailored to reason across multiple ASATs.

2.2 Research Questions

The goal of this chapter is to *increase our understanding of how static analysis tools are used in the real-world.* To that end, we study a large collection of OSS projects from both statically (Java) and dynamically typed languages (JavaScript, Ruby, and Python) in professional and non-professional popular OSS settings.

In pursuing our goal, we must first establish a baseline of how widely-used static analysis is in these projects. Hence, in our first research question, we ask:

RQ I.1 How common is the use of ASATs in practice?

We refine the research question into three sub-research questions that we investigate:

RQ I.1.1 What is the prevalence of ASATs?

RQ I.1.2 How common is the simultaneous use of multiple ASATs?

RQ I.1.3 Do projects enforce ASAT use?

Having gained insight into how widespread the use of ASATs is through manual analysis, we set out to study how a large corpus of projects use ASATs by automatically investigating the ASAT configuration files in their repositories:

RQ I.2 How are ASATs configured?

The answer to this this research question can be important for the creators of ASATs and their users: Coming up with sensible defaults for software is a hard problem [107]. A large-scale study of their user base could help ASAT developers uncover if their defaults generally fit the tool's use in practice so that users spend less time configuring their ASATs. To this end, we want to gain insight specifically into the following sub-research questions:

RQ I.2.1 What type of warnings are explicitly enabled?

RQ I.2.2 What type of warnings are explicitly disabled?

RQ I.2.3 How well do default configurations reflect real-world configurations?

RQ I.2.4 How prevalent are custom rules in the OSS configurations?

Finally, in order to understand which role ASATs take in the development process, and if and how their configurations files change over a project's lifetime, we ask:

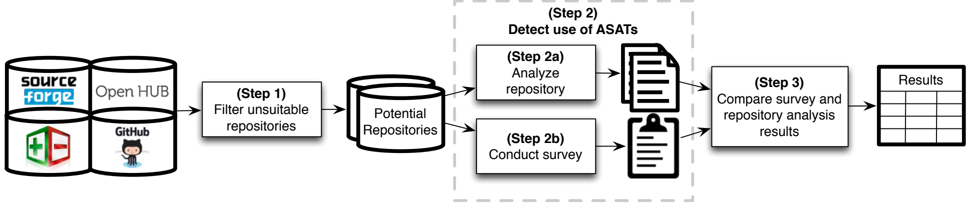


Figure 2.1: An overview of the study design for the prevalence analysis.

RQ I.3 How does the use of ASATs evolve?

Specifically, we answer the following sub-RQs:

RQ I.3.1 How often do ASAT configurations change?

RQ I.3.2 How much do ASAT configurations change?

RQ I.3.3 When do ASAT configurations change?

2.3 Prevalence Analysis (RQ I.1)

In this section, we address the question of how wide-spread the use of static analysis is in popular OSS Projects.

2.3.1 Methodology

To answer this question, we followed the study design depicted in Figure 2.1. We started by examining the four popular OSS project hosting platforms GitHub, OpenHub, SourceForge, and Gitorious (Step 1) in December 2014. We also considered other sources, primarily other code hosting services, but found them unsuitable: Some lacked representative projects (for example, on GitLab [108], the most popular repositories belonged to the GitLab company itself), others provided no means of ranking projects by their popularity (for example, the now-defunct Google Code [109]).

Proportional to the number of projects that are hosted on each platform [110], we selected the 100 most popular repositories on GitHub (ranked by number of stars), 20 on Gitorious (ranked by development activity), and 10 from both SourceForge (ranked by number of downloads) and OpenHub (ranked by number of users). In contrast to the other three, OpenHub is not a forge, but a "public directory of free and open source software," which includes links to the project's actual repository. OpenHub's overall popularity ranking was only available for the 10 most popular projects. After eliminating duplicates and non-software repositories, we ended up with 122 unique projects to analyze.

Using a mixed methods approach, we investigated their use of ASATs in two distinct ways (Step 2). On the one hand, in a manual analysis of the projects' websites, contribution guidelines and ASAT configuration files in repositories, we investigated whether and how projects documented their ASAT use (Step 2a). On the other hand, we sent out a short survey to contributors of the same 122 projects, asking which static analysis tools they are using, when they are using them, if it is a necessary precondition to check code before it can enter the main project repository, and whether ASATs are an integral part of their

Table 2.1: Prevalence of ASATs according to our **Repository Analysis**.

Source	Projects	Use 1 ASAT	Use > 1 ASATs
GitHub	83	34%	30%
OpenHub	9	67%	22%
SourceForge	10	30%	0%
Gitorious	20	30%	5%
Total	122	36%	23%

Table 2.2: Prevalence of ASATs according to our **survey**.

Source	Projects	Use 1 ASAT	Use > 1 ASATs	Enforce Use
GitHub	19	36%	32%	42%
OpenHub	1	0%	0%	0%
SourceForge	3	34%	66%	0%
Gitorious	10	30%	40%	30%
Other †	3	100%	66%	33%
Total	36	41%	36%	36%

† Replied for their project B as a reaction to our survey in a mailing list of another project A (unrelated to B).

workflow (Step 2b). In order to maximize the number of responses, we sent this question to the projects' mailing lists, newsgroups, or fora, and contacted the two top-committers. We also explicitly lowered the barrier to entry of the survey by embracing a discussion-style answer of developers directly to our informal email [111]. Finally, in Step 3, we compared the results that we had collected using Steps 2a and 2b.

2.3.2 Results

In this section, we introduce the results from the manual repository and website analysis, then describe the corresponding results from the surveys, and finally compare them.

Repository Analysis. Table 2.1 presents an overview of the results from analyzing the information in project repositories and websites regarding ASATs for RQs I.1.1–1.3. Overall, our results stem from analyzing 122 projects (see Bholanath's thesis [111, Appendix A] for the complete list). Most of them (36%+23%=59%) either mention the use of ASATs in their official project documentation, or their repository contains ASAT configuration files or their build processes specify explicit dependencies on ASATs. Of those projects, 28 use multiple ASATs. Examining the project sources separately, 53 out of 83 GitHub projects use ASATs and 25 of those use multiple ASATs. Only one OpenHub project does not use ASATs, and 2 of the other 8 projects use multiple ASATs. ASATs are not popular among our SourceForge projects, with only three of them adopting ASATs, all of which use a single ASAT. Finally, 7 out of 20 Gitorious projects use ASATs, but only one of them uses

multiple ASATs.
Survey. We received responses from 36 projects, achieving a relatively high response rate for surveys of 30%. Table 2.2 shows the corresponding results from the survey. The last column displays the percentage of projects that use the results of these tools as one of the factors to decide whether a code contribution should be integrated into the project repository. This displays information that we could not always obtain from a repository analysis. Overall, we observe that ASATs are used by 41% (Table 2.1) + 36% (Table 2.2) = 77% of projects who answered the survey. Most respondents state that ASATs are only sporadically used by developers when they believe that a code change warrants ASAT use.

Concerning RQ I.1.3, we observe that a slight majority of the projects that use ASATs, 15 out of 28, rely on a single tool. Five projects use more than two ASATs, with no project using more than three comparable ASATs. Abilian [112] is the only project that uses more than three ASATs, but for three languages (JavaScript, CSS, and Python). All other projects only use a single ASAT or multiple ASATs that check for defects in the same language. Slightly less than half of the projects that use ASATs (13 out of 28) place a strict ASAT-regression policy on new code. This means that code that is submitted for review or integrated into the project repository must not introduce new ASAT warnings.
Comparison. Using a mixed methods approach to evaluate the validity of our results [66], we compared the detailed results that are summarized in Tables 2.1 and 2.2 for all 36 projects for which we had both sources available. For close to 20% of the projects that responded to our questionnaire, the repository analysis results deviate in some way from the survey responses, which we consider to be the ground-truth. In three cases, the repository analysis shows that ASATs are used, while in reality, the project does not use any ASATs. A reason for this might be that the information that we gathered from the repository or website might be outdated. For example, the Bash project [113] mentioned that they previously used Coverity [114], for which traces can still be found in existing sources. For two other projects, two tools are found in the project information, while respondents only note that one of them is in use. Moreover, two projects use a different ASAT than reported. Furthermore, there are seven projects that used more ASATs than the repository analysis indicated and a single project that used ASATs even though the repository analysis showed otherwise.

2.4 General Defect Classification (GDC)

When we reason about multiple ASATs to answer RQs I.2 and I.3, we are confronted with the problem that each tool provides a plethora of different individual rules (checks), often without an explicit ordering scheme or a topology. If we want to derive meaningful results from a comparison of multiple ASATs, we therefore need to design a common, more general classification scheme to allow us to abstract over the tool-specifics. To that end, we propose the GDC, which is based on the *code review defect classification* [27], which lent itself to an adoption of ASAT warnings because it categorizes "human static analysis" (i.e., code reviews), similar to "automatic static analysis" warnings. One useful property of this defect classification is that it can be used to categorize not only defects, but also warnings (or, review suggestions), and actual code changes, as we have shown before [27].

Figure 2.2 depicts the two different high-level categories of the GDC, maintainability and functional defects. It also shows the 16 second-level defect categories (7 functional,

2

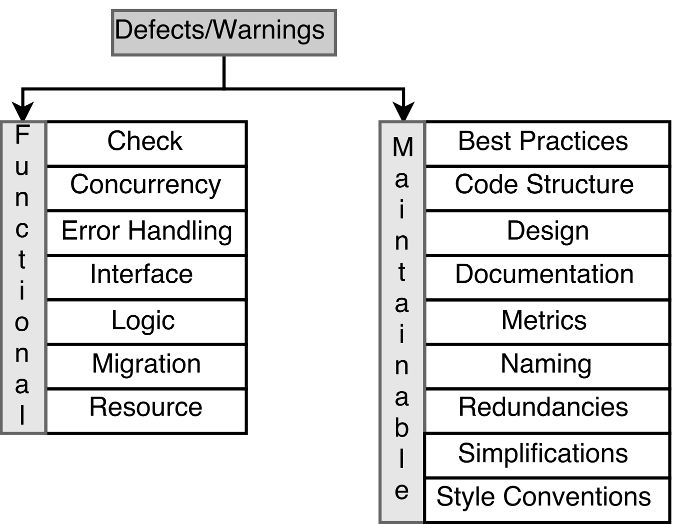


Figure 2.2: Top- and second-level categories of the GDC.

9 maintainability). Similar to our prior work [27], there are two top-levels categories, functional and maintainability. Each of them is refined into a set of sub-categories that further characterize the warning. As one example of a new category, “Metric” pertains to warnings that “measure a certain attribute of the code,” like the “NestedIfDepth” warning in CHECKSTYLE.

In a grounded-theory-driven approach, the first two authors separately browsed through all available FINDBUGS checks and tried to classify them into fitting groups, using the code review classification scheme as a blueprint [27]. Wherever we could not find a suitable existing category, we introduced a new one and sorted it into the topology. Upon completion of this task, the authors met and compared their adopted classifications, distilling a common first draft of the GDC. After this, we mapped the traditionally more maintainability-oriented CHECKSTYLE warnings into this preliminary ASAT warning topology. In this round, we soon reached saturation and only introduced two new categories under the maintainability sub-level.

To stimulate future research, we distribute our GDC classification along with detailed explanations on the error types and our manual mapping of all 2,385 checks of the nine ASATs to their corresponding GDC categories freely.[1]

2.5 Configuration & Evolution (RQ I.2, RQ I.3)

In this section, we describe the design and results of our studies that address RQs I.2 and I.3.

[1]http://dx.doi.org/10.6084/m9.figshare.1603419

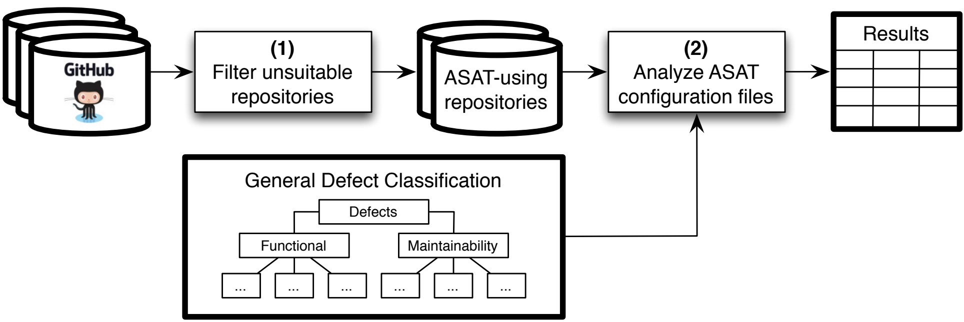


Figure 2.3: An overview of the study design for the ASAT configuration analysis.

2

2.5.1 Study Design

Figure 2.3 depicts our high-level study design for RQs I.2 and I.3. To facilitate the technical part of our analysis, we first decided to only consider projects that are hosted on GitHub (1). Having chosen a selection of nine ASATs, we then developed two tools: ASAT-Configuration-Analyzer[2] for RQ I.2 and the ASAT-History-Analyzer[3] for RQ I.3. Before we could use them, however, we had to crawl GitHub for the occurrence of any of the supported ASAT configuration files and store a URL at which we can retrieve the content of the file (1). The tools then receive a list of URLs to configuration files, which it downloads and parses, applying the mapping of the individual tool checks to the GDC from Section 2.4 (2). The results are classified distributions of warnings that capture how developers configure their ASATs on a more abstract level than the individual tools would allow. We answer our research questions on the basis of these distributions.

2.5.2 Methods

In this section, we describe our study methodology.

Selection of ASATs (1). We placed some restrictions on the ASATs that we could use. First, an ASAT has to be configurable. If an ASAT is not configurable, then no study regarding its use is necessary. We can simply conclude that all developers use the ASAT in the same manner. Furthermore, if an ASAT is configurable, it needs to store its configuration in a separate file (and not, for example, via command line arguments). Finally, the configuration file needs to be parsable. In practice, this means that the configuration needs to be in a machine-readable format such as XML, JSON, or even a custom key-value pairing.

We used the ASATs that we encountered for RQ I.1 (see Section 2.3) as a starting point. We expanded our search with search engines and programming support sites such as Stack Overflow [115]. Table 2.3 lists the nine tools which fit our criteria. Most tools use standard formats to store their configuration. Two tools, JSL and Pylint, use key-value pairs in plain text format. FindBugs is a peculiar case. The tool uses XML files to either exclude or include rules in a specific class, file, or package. However, whether an element is an inclusion or an exclusion of a rule is specified via command line arguments. Thus, we

[2]https://github.com/rbholanath/ASAT-Configuration-Analyzer
[3]https://github.com/rbholanath/ASAT-History-Analyzer

Table 2.3: Description of the ASATs for RQ I.2 and I.3.

Tool	Language	Format	Extendable	Released	# of Rules
CHECKSTYLE [116]	Java	XML	Yes	2001	179
FINDBUGS [117]	Java	Text	Yes	2003	160
PMD [118]	Java	XML	Yes	2002	330
ESLINT [119]	JavaScript	JSON	Yes	2013	157
JSCS [120]	JavaScript	JSON	Yes	2013	116
JSHINT [121]	JavaScript	JSON	No	2011	253
JSL [122]	JavaScript	Text	No	2005	63
PYLINT [123]	Python	Text	Yes	2006	390
RUBOCOP [124]	Ruby	YAML	Yes	2012	221

could not determine this in a consistent way. Instead, we used the configuration files of the FINDBUGS Eclipse plugin. This plugin also stores its configuration in plain text key-value pairs.

One factor that could influence how developers configure their tools is what type of defects a tool focuses on. For the Java tools, CHECKSTYLE focuses primarily on coding style, FINDBUGS on functional defects, and PMD tries to find both types. For the JavaScript tools, JSCS focuses on coding style rules, while both JSHINT and ESLINT try to find all types of defects. JSHINT will refocus in an upcoming major release to functional defects and has marked many rules as deprecated in preparation for the removal of these coding style rules [125]. This might already have affected the configurations of developers, if they stopped using the deprecated rules in preparation for the change. JSL, PYLINT, and RUBOCOP do not state a particular focus on a specific subset of defects. However, RUBOCOP seems to favor checking for coding style issues, as made evident by the fact that most of their rules are classified by the tool itself as belonging to the *Style* category.

Analyzing Configuration Files (2). Developers can configure most ASATs to fit their specific needs through a configuration file. For RQs I.2 and I.3, we study how developers use ASATs through them as a proxy. In an ASAT configuration, developers can enable the rules that check for defects that they consider important, and disable rules they do not deem important (e.g., perhaps because of a high false positives rate). Without a configuration file, developers rely on the default, which might not align with their specific needs. The contents of an ASAT configuration file are hence an important indicator of how developers use ASATs to check for defects in their code and how well the tool's default settings reflect its use. A version-controlled ASAT configuration is crucial for collaboration because it enforces consistent static checks across developers.

For RQ I.2.1 and RQ I.2.2, we were interested in the distribution of the rules that are explicitly activated by developers and likewise those that are disabled by developers. This indicates which types of warnings are considered important by developers, and conversely, which types of warnings they avoid. To mitigate the influence of the set of possible ASAT rules, we normalized these distributions according to the number of rules in a category. To see why this is important, consider a hypothetical tool with just two defect categories, A and B, where A has one rule and B has two. If the developers enable the rule in category

A once and each of the rules in category *B*, then a uniform distribution of defects would show *B* as being twice as actively-enabled as *A*. However, we can see that each individual rule in *A* and *B* was enabled once. The results in this normalized form allow us to study the relations between a category with a large number of rules and another with just one or two rules.

For RQ I.2.3, we were interested to see if and how the configurations of developers deviated from the defaults, as these are indications of whether the default configuration accurately reflects the wishes of ASAT users. A developer can deviate from the default configuration in three ways: 1) Disable a rule that was enabled by default, 2) Enable a rule that was disabled by default, or 3) Reconfigure a rule.

Not all rules can be reconfigured. An example of a configurable rule is the "NestedIfDepth," for which the depth threshold can be customized via a simple integer value. Reconfiguring a rule indicates that developers want to check for this convention, but do not agree with the default convention as specified by the creators of the tool. We assume that a rule is reconfigured when a developer includes an enabled default rule in his own configuration.

To see if developers deviate from the default, we simply computed what percentage of configuration files included one or more deviations for a default rule. To examine how developers deviate from the default configurations, we computed, for each rule, how many configuration files included a particular type of deviation for that rule.

For RQ I.2.4, we determined the prevalence of customized warning rules in comparison to the built-in rules of a tool. We consider a rule to be custom-made if it was not included as a built-in rule in a recent version of the ASAT. For each tool, this can indicate whether the developers consider the tool to be incomplete, which might result in developers writing custom rules to find these defects. Generally, this can be an indication of whether current ASATs can adequately cover the defects that developers wish to find.

Analyzing Configuration Evolution (2). For each identified configuration file from (1), we then performed an analysis of its evolution over time. The first metric, for RQ I.3.1, is simply how often a file was changed. This tells us if developers have a need to adapt their ASAT configuration, either because the ASAT was updated or because of changing needs among developers. For RQ I.3.2, we calculated the total number of line changes in a file. We defined this as the difference between the number of lines added and deleted in a single change. If this number is zero, it likely means that there are only lines modified, which count as both an addition and a deletion in the information of a change. We did not compute more fine-grained measures, such as an edit distance, because of the excessive computing load for all changes of the more than 160,000 configuration files and its relatively small expected information gain.

2.5.3 Study Objects

After selecting the ASATs to study, we needed to retrieve configuration files for every tool. We expected to find enough configuration files on GitHub. However, to further augment the study, we also collected data from Krugle [126] and OpenHub. To eliminate possible duplicates, we excluded OpenHub results which hosted their code on GitHub.

Table 2.4 details the number of configuration files split per ASAT and hosting site. As projects typically have one configuration, the numbers are a good estimator for the

Table 2.4: Configuration files for each ASAT, grouped by source.

Tool	GitHub	OpenHub	Krugle	Total
Checkstyle	16,271	2,492	22	18,785
FindBugs	1,575	514	1	2,090
PMD	5,562	1,888	8	7,458
ESLint	4,427	5	3	4,435
JSCS	11,656	20	1	11,677
JSHint	105,619	3,086	65	108,770
JSL	862	0	0	862
Pylint	3,941	123	7	4,071
RuboCop	10,063	0	3	10,066
Total	159,976	8,128	110	168,214

number of different projects. We identify JSHint as the most wideley-used ASAT among our selection. The number of added files from Krugle is minimal for all tools. Moreover, for some tools, there were more configuration files hosted on GitHub than we could access due to limitations in GitHub's search (see Section 2.6.2).

2.5.4 Results

In this section, we detail the results to RQs I.2 and I.3.

Results to RQ I.2. RQ I.2.1 and 2.2 are concerned with the warning rules that developers enable and disable respectively in their configurations. As explained in Section 2.5.2, we normalized the distribution of the enabled and disabled warning rules according to the number of rules in a category. Figure 2.4 details these normalized results for every tool. Every one of the 9 bars displays the percentage of normalized rules that belong to a specific category in our classification. Due to this, the differences in Figure 2.4 from a uniform distribution are due to developers over- or under-proportionally enabling or disabling rules in this category. As an example, almost 10% of the normalized rules that are enabled in FindBugs configurations belong to the Check category. The figures also allow us to identify categories that are outliers for a specific tool. For instance, the Metric and Migration categories contain a large percentage of the enabled rules for RuboCop. In both the enabled and disabled distributions, some tools show categories with no enabled or disabled rules.

For RQ I.2.3, we calculated how many configurations deviated from the default. The second column of Table 2.5 shows how many configuration files changed one or more default rules, i.e., disabled a rule that was enabled in the default configuration or vice versa. The third column shows how many files did not change a default rule, but reconfigured a default rule. Blank spaces indicate that the tool does not allow individual rules to be reconfigured. The fourth column lists the percentage of configuration files that do not contain a deviation for any default rule.

With Table 2.5 as the basis, we also assessed in three ways how developers deviate from a default configuration. First, for all rules that are enabled in default configurations,

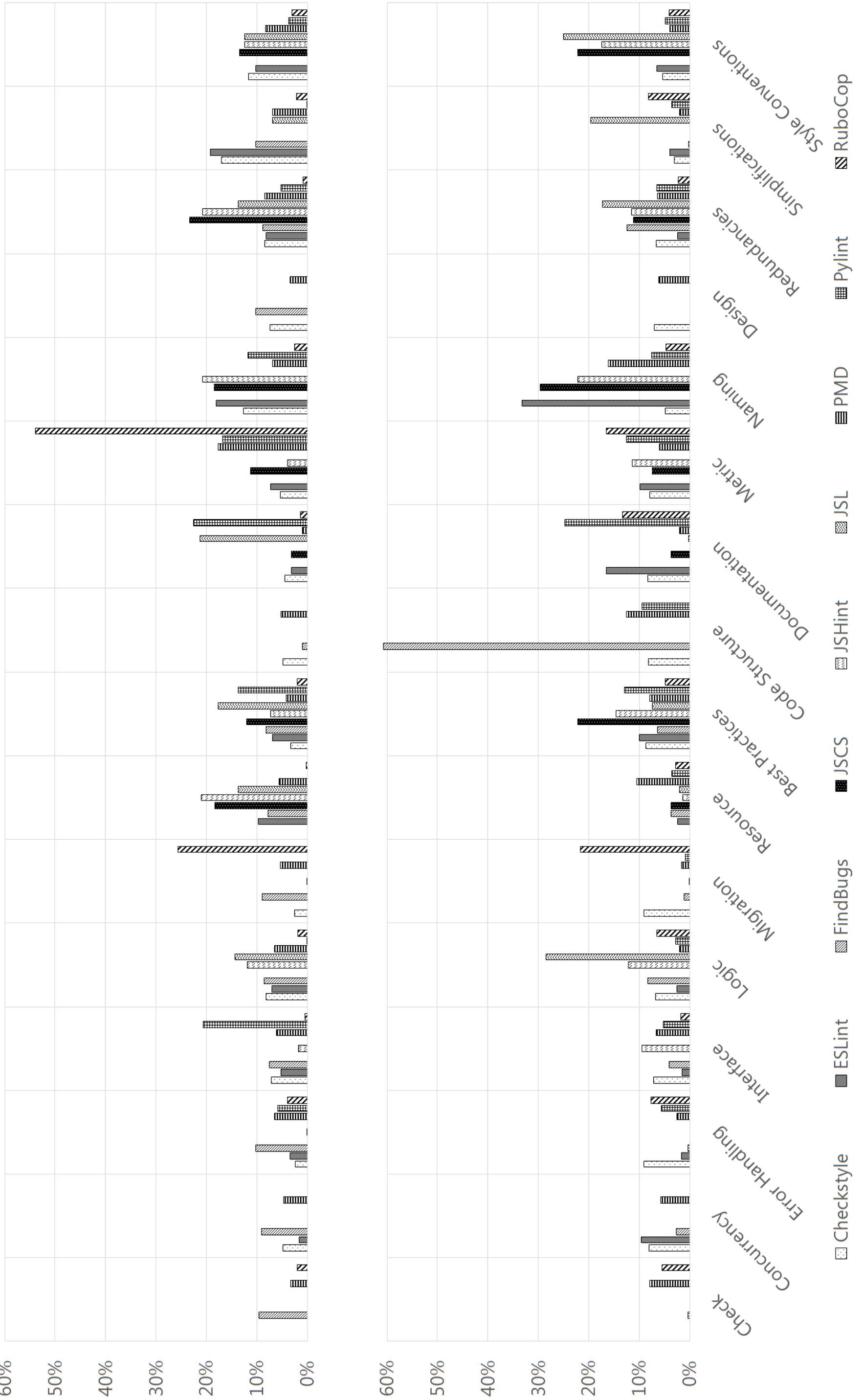


Figure 2.4: Normalized average means of enabled (top) and disabled (bottom) checks per ASAT for all 16 second-level GDC categories.

we calculated how many developers disabled them. Subsequently, for every rule that was turned off by default, we calculated in how many configurations that rule was enabled. Finally, for every rule that was enabled in the default configuration and which could be configured, we calculated how many configurations possibly reconfigured them.

Our results show that there is wide agreement with the default rules that ASAT providers ship: Only for the tools ESLint (2% of default rules affected) and JSHint (10%) did more than 50% of configuration files deviate from the default by reconfiguring a subset of rule defaults. For FindBugs and JSHint more than 50% of developers enabled 2 and 5 default-disabled rules, respectively.

For RQ I.2.4, we calculated the percentage of custom rules in the configuration of developers. Table 2.6 shows the results. Custom rules never account for more than 5% of all of the enabled rules of a tool. For 3 out of 8 ASATs, this percentage is even lower than 1%. We omit JSL from these results because JSL does not permit custom rules.

Results to RQ I.3. Figure 2.5 shows the results for RQ I.3.1 regarding how often and how profoundly configuration files change. A little over 80% of all configuration files are never changed after their creation. The range in the chart represents 99.5% of the total data. Less than 10% of all files are changed just once and less than 5% twice. The maximum number of times that a configuration file was changed is 248, for a Checkstyle configuration.

For the 19% of configuration files that were changed after their initial creation, we analyzed each change of every file to determine the size of the change. The distribution in Figure 2.6 shows the results to RQ I.3.2. The total number of changes is zero for more

Table 2.5: Summary of rule changes from default configurations.

Tool	Changed	Reconfigured	No Deviations	Total
ESLint	80.5%	5.7%	13.8%	4,274
FindBugs	93.0%	—	7.0%	2,057
JSHint	89.6%	0.7%	9.7%	104,914
JSL	94.6%	—	5.4%	848
Pylint	53.3%	—	46.7%	3,951
RuboCop	79.1%	3.2%	17.7%	9,579

Table 2.6: Average mean of custom rules in ASAT configurations.

Tool	Percentage of Custom Rules
Checkstyle	0.2%
ESLint	4.1%
FindBugs	1.3%
JSCS	4.7%
JSHint	0.1%
PMD	2.9%
Pylint	1.1%
RuboCop	0.9%

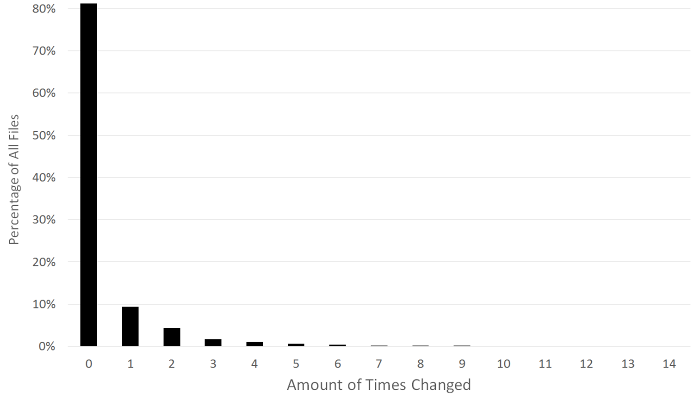


Figure 2.5: Number of changes to an ASAT configuration (median 0, mean 0.5).

than 25% of all files. This means that either lines were only modified "in-place," or that there were as many completely new lines added as deleted. Furthermore, there is a greater chance that a change has more additions than deletions. The range in the chart captures more than 90% of the data. The rest of the data is spread out from -1,126 to 2,055 total changes.

The distribution in Figure 2.7 shows the results to RQ I.3.3. We see that 18% of the changes are made on the same day that the file is created and 33.5% of changes are made within the first week. The tail of the data is quite extensive, as the range shown in the chart covers just over 65% of the data. However, no date more than 15 days after the creation of the file individually represents more than 1% of all changes. The maximum is 11.5 years for a Checkstyle configuration.

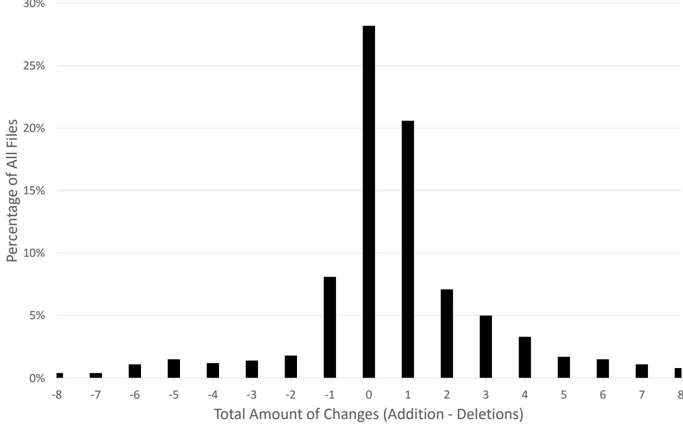


Figure 2.6: Distribution of size of line changes after initial configuration (median: 1, mean: 1.64).

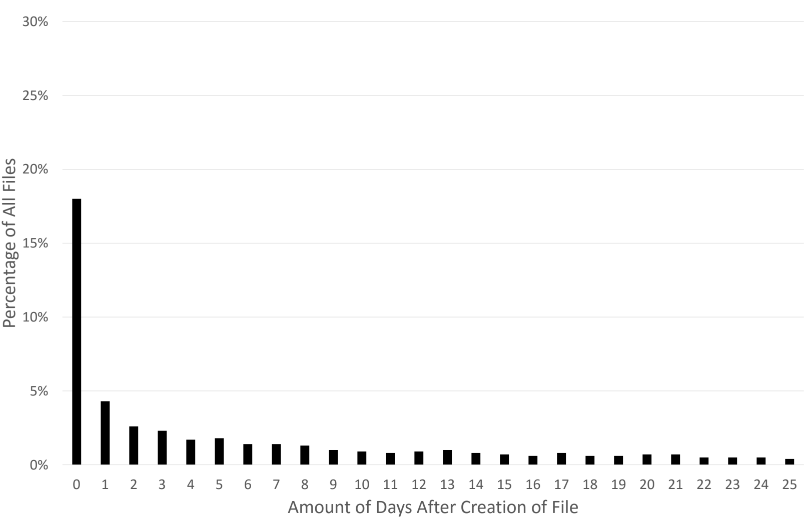


Figure 2.7: Distribution of changes per days after initial configuration (median: 32 days, mean: 151).

2.6 Discussion

In this section, we discuss our results and possible threats to the validity of our conclusions.

2.6.1 Results

For RQ I.1.1, we found that the percentage of projects using ASATs according to the survey is higher than that of our repository analysis. Excluding projects that are present in both sets, 77% of the respondents of our survey note that ASATs are used compared to 52% of project resources. It seems highly likely that the respondents to our survey were more inclined to use ASATs, possibly explaining the 25%-point difference.

> The results of a large mining repository analysis give a useful approximation of the real ASAT use of OSS projects. It might be inaccurate on a single project basis.

However, the results from the survey also showed that the majority of projects that use ASATs (15 out of 28) only run these tools sporadically and without enforcing them. Researchers should avoid using data that is solely collected from project repositories and documentation to draw conclusions about ASAT use. This highlights the need to analyze multiple data sources in empirical software engineering [66, 127].

Our manual repository investigation showed that less than 59% of projects use ASATs in various levels of strictness for RQ I.1.2. These results seem to contradict prior results [89, 128, 129], which claim that ASATs have not yet achieved significant use among software projects.

> ASATs are common, but not ubiquitous in popular OSS.

Table 2.4 shows that our ASAT selection contained six times more JavaScript projects that used an ASAT than such Java projects. This is against our expectation since there are only three times more JavaScript than Java projects on GitHub. Before drawing further conclusions, we need to evaluate this initial finding on a larger set of dynamic and static languages.

> Projects in a dynamically-typed language such as JavaScript might require or benefit more from ASAT use than projects in a statically-typed language such as Java.

The questionnaire also showed that the way in which ASATs are used varies. 64% of projects use ASATs sporadically and without attaching any consequences to the warning results.

> Few projects have ASATs tightly integrated into their workflows and even fewer projects mandate that the codebase should be ASAT-warning free.

Past research has suggested that an important factor of improving the adoption of ASATs was to make this integration as easy and seamless as possible [89]. For instance, Coverity provides both GitHub and Travis CI [130] integration [131, 132], making it easy to integrate ASATs into a feedback-driven development workflow (see Chapter 1).

> In order to fully benefit from ASATs, projects should include them into their standard workflow, for example as part of their continuous integration processes.

Concerning RQ I.1.3, we observed that most projects use one ASAT. This is in spite of the fact that the use of multiple ASATs can provide a large increase in defect detection capabilities [83, 90–92]. Developers might be unaware of these benefits or an overload of warning messages generated by multiple ASATs might cause developers to avoid them [89].

For RQ I.2.1, we observed from Figure 2.4 (top graph) that 65% of all enabled rules belong to the GDC maintainability defect category. The other 35% of rules belong to a functional defect category. A reason for this might be that, ASATs perform poorly at finding functional defects [92, 97–99]. Ayewah [96] and Wagner [97] argue that the reason could be that ASATs do not know what code is intended to do, which is crucial if one wants to find functional defects. If developers notice the poor performance of these functional defect rules they might place less importance on them and subsequently leave them out of their configurations.

Concerning RQ I.2.2, from Figure 2.4 (bottom graph) we observe that 75% of all the disabled rules are maintainability defects. However, the ratio of maintainability defects

to functional defects is not significantly larger for the rules that developers disable than it is for those rules that developers enable. Even though the ability of ASATs to find functional defects is limited [92, 97–99], developers do not widely disable these rules. A potential reason for this might be that these rules do not emit a lot of false positives. On the contrary, a rule that never emitted a warning might not be worth disabling, as it might still find a defect in the future.

The majority of actively enabled and disabled rules are maintainability-related.

As we have only compared the choices that developers make in their configuration files explicitly, this high-level observation is not a contradiction. To investigate it in more depth, we need a study that also takes into account the implicit defaults of configurations.

On a lower level, Figure 2.4 shows some outliers for individual tools. For instance, regarding enabled rules, the *Metric* category for RuboCop and the *Logic* category for Pylint stand out. For disabled rules, the FindBugs *Code Structure* category and the RuboCop *Migration* category are noticable outliers. These outliers indicate that, for a single tool or programming language, developers sometimes consider a specific category less or more important than developers using other ASATs or languages.

For the results regarding RQ I.2.3, Table 2.5 shows that, for all tools, less than half of all configurations do not change or reconfigure any rule from the default configuration. For 5 out of 6 tools, this percentage is even lower than 20% and for 3 out of 6 tools it is less than 10%.

Most configurations change or reconfigure rules from the default configuration, but typically only one rule.

The results described in Section 2.5.4, and Table 2.5 and Table 2.6 in particular, indicate that there are few rules that a noticeable percentage of all developers change or reconfigure. Figure 2.4 could suggest improvement opportunities in the default configurations of ASATs. For the enabled rules, 5 out of 7 tools have zero default rules that are disabled by developers in more than 25% of all configuration files. Moreover, less than 5% of the rules for the other two tools are disabled more than 25% of the time. For the rules that are disabled by default, 3 out of 6 tools do not have any rules that are turned back on by more than 25% of all developers. The other three tools have a higher number of such rules. Most striking are the results for FindBugs. Even though there are just eight rules that are disabled by default, the results show that the default configuration should probably enable rather than disable some of those rules. Regarding the reconfigurable rules, the percentage of rules that are potentially reconfigured by developers are low among all three tools. However, both ESLint and JSHint still have rules that pass the 50% mark. The creators of these tools should therefore consider changing the default settings.

Developers only widely disagree with few rules in default configurations.

Finally, regarding RQ I.2.4, our results show that custom rules do not comprise a sizable segment of all rules, amounting to less than 5% for all tools. This can indicate that developers do consider the ASATs to be complete, in the sense that they need not create custom rules to check for defects that are not included in the built-in rule set. Nevertheless, this could also be an unwillingness to create custom rules, with developers manually checking for those rules they consider to be missing in the ASATs that they use.

> Custom rules comprise less than 5% of all rules that are used by developers.

Regarding RQ I.3.1, the results show that the use of ASATs is relatively stagnant (i.e., does not evolve). Over 80% of all the configuration files that we analyzed are created and then used as-is for the remainder of the project's lifetime to date. Moreover, only 5% of all configuration files are changed more than twice and less than 2% are altered more than five times.

> Most configuration files never change.

Looking at only the files that are changed, the results for RQ I.3.2 show that, for most files, the total number of changed lines lies within a reduction of five lines to an increase of five lines. Furthermore, more than 28% of all files have an equal number of added and deleted lines, indicating that there were likely only modified lines.

> Most changes to configuration files are small in size.

The results for RQ I.3.3 show that a configuration files is most likely to be changed on the same day that it was created. Looking ahead one week, we see that slightly over a third of all changes were made in this time span. Going even further, almost half of all changes are made within a month after a file's creation. Thus, we observe that developers that make changes to their configuration files do not only do so in the period where they are still getting used to the ASAT. Assuming that this period lasts a week, or surely no longer than a month, at least 50% of all changes are made after the ASAT was used for a lengthy amount of time.

> A third of all changes happen in the first week after the creation of an ASAT configuration.

2.6.2 Threats to Validity

In this section, we explain which threats affect the validity of our study and show how we mitigated them.

Internal Validity.
Since most study points stem from GitHub (see Table 2.4), this might bias our results. To minimize the bias, we looked for all code hosting services and search engines that allowed us to find ASAT configuration files. Hence, our bias towards GitHub might simply be reflective of its current popularity among OSS projects.

There might be errors in our measurements due to the use of our analysis tools. We verified that the tools worked as expected on small, manually curated samples and through automated tests. Moreover, we programmed our tool defensively, that is, the tool skips those configuration files that do not conform to a strict specification. To mitigate this risk further, we open-sourced our tools.

GitHub's search only retrieves 1,000 hits. We worked around it by strategically boxing and modifying its file size parameter in one-byte increments. However, this was too coarse-grained for some searches. As a result, we could not retrieve a few hundred configuration files for most tools, and about 220,000 for JSHint. Our sample size of over one third of the total JSHint population is still significant.

For RQ I.2.3, we assumed that the current default settings still applied when the ASATs were initially adopted by the studied repositories. If the default configuration changed significantly over time, our results might be inaccurate. However, manual inspection of a few projects showed that the default typically evolves gracefully, adding new options, but not changing existing ones.

External Validity.
This study only considers the configurations of ASATs from OSS projects. As such, its generalizability towards closed-source projects might be limited.

Our study targets nine ASATs and four programming languages, representing a diverse set of tools (see Table 2.4). Therefore, we expect those results that abstracted over all the tools and presented a general view of the studied ASATs to further generalize over ASATs outside of this study as well. However, replication studies are needed to confirm this.

2.7 Tool Construction UAV

In this section, we describe the tool UAV, which can show warnings from multiple ASATs simultaneously to help promote the benefits of using multiple, complementary ASATs simultaneously and thus alleviate some of the hindrances to a wider ASAT adoption unveiled in our case study.

2.7.1 Introduction

Developers currently have little guidance which ASATs to choose and combine for a project. They lack a tool that allows them to explore *which* warnings a *certain* ASAT emits *where* in the project, and whether there are *overlaps* with existing ASATs. Currently, developers and researchers can only run ASATs individually and then compare their output, which is both tedious and leaves many features to be desired. As a result, many projects still only employ one ASAT with practically no customization and never explore the possibility of combining multiple ASATs to their benefit Section 2.6.

To address this issue, we have created UAV, the Unified ASAT visualizer. UAV can run multiple ASATs and facilitates comparing them by unifying their different warning

typologies and representing all warnings in one interactive treemap visualization. For researchers, UAV offers a flexible means to analyze the different types of warnings generated by multiple ASATs. For software developers, our tool gives insight into the warning distribution in their Java projects. After locating a specific class full of warnings, developers can seamlessly navigate to the source code view where the relevant warnings will be highlighted. UAV can also support ASAT tool creators themselves by helping them sharpen the focus of their tools: They can compare the warning types their tool detects to its competition and thus differentiate themselves better. In the remainder of this paper, we describe UAV from an end-user as well as a technical perspective.

2.7.2 User Story

Scott and his team of software engineers at XYZ Inc. are developing a revolutionary new search website. They decided to use multiple ASATs to ensure a basic level of code quality, including Checkstyle and FindBugs. After a few weeks of development, Scott checks all warnings. To his surprise, the ASATs report a list of over a thousand warnings on the relatively new project. Scott wants to address the warnings in an efficient manner, but has no idea where to start. He is discouraged by the fact that he cannot get an overview of how the warnings are distributed across the system's components. For example, warnings related to the search subsystem would take precedence over warnings in the user interface (UI) components of the new search website. Scott knows that working through the lengthy list of warnings one at a time will be extremely time-consuming, but sees no other option. Working through the list Scott repeatedly notices overlaps between warnings from different ASATs, albeit under slightly different names. For example, for the method `AdvanceState`, Checkstyle and FindBugs emit the overlapping warnings `MethodName` and `NM_METHOD_NAMING_CONVENTION`. Scott realizes that he is losing significant time on similar issues. Moreover, he has no way to exclude warnings which are irrelevant to his team. Scott wonders: Isn't there a tool which provides me with ...

- an overview of where in the system warnings are concentrated?
- an overview of warnings which have the highest priority?
- a way to filter irrelevant warnings?
- a way to filter overlaps in the warnings from multiple ASATs?

2.7.3 Related Work

In this section, we give an overview of literature and tools that are related to UAV.

Literature

Many ASATs differ in the type of defects they detect. However, even when tools focus on uncovering the same category of defect type, the variance in the concrete warnings they emit and their naming is still very large [60]. This indicates that using several ASATs has benefits over using a single ASAT. Using multiple ASATs can be time consuming, however, arbitraging a single warning can take up to eight minutes on average [86]. Moreover, ASATs have been observed to generate a large number of (false) warnings, about 40 per 1,000 lines of code in the Faultbench benchmark [88]. With UAV, developers and

2

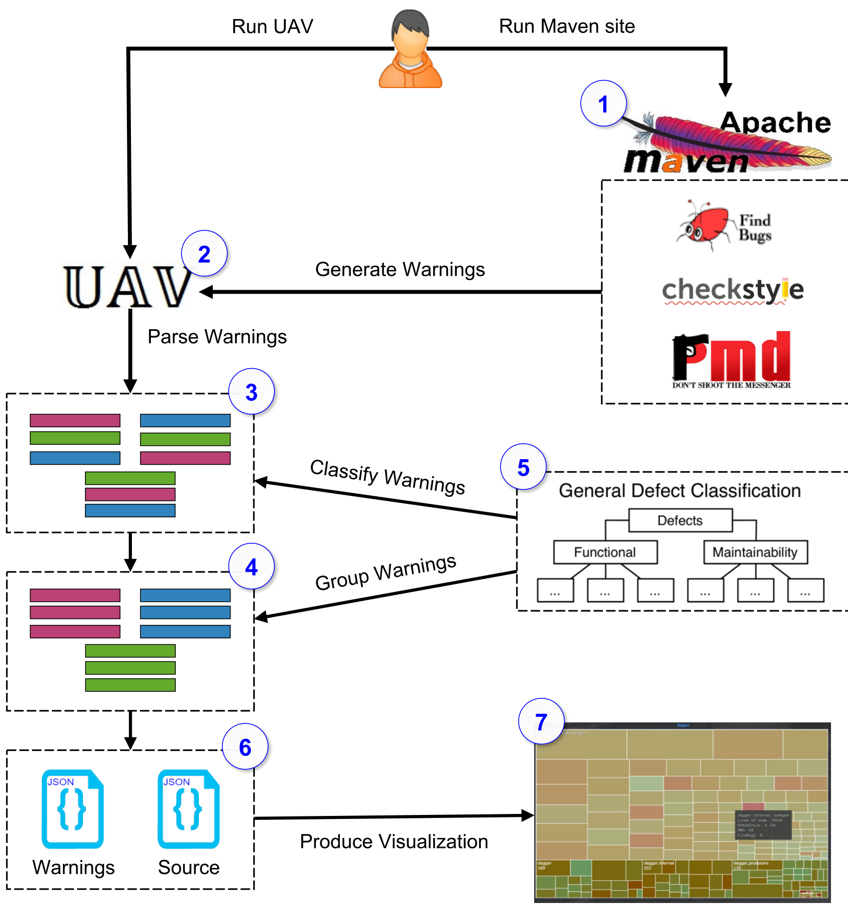


Figure 2.8: Workflow of UAV.

researchers can visually assess this rich and plentiful torrent of warnings for the potential benefits of combining multiple ASATs. It enables developers to make an informed decision on whether the added findings and their type justify the inclusion of another ASAT into their tool chain. Researchers have long performed comparative studies with multiple ASATs and other quality assurance techniques such as code review, for example Wagner et al. in 2005 [91] and Panichella et al. in 2015 [133]. However, they lacked a tool that allows them to visually compare the location and defect types of different ASATs on concrete real-world projects. UAV closes this gap.

User Workflow

UAV offers a visual way of exploring which packages or classes are particularly affected by ASAT warnings. By contrast, existing research has tackled the problem of how to deal with a flood of warnings mainly by prioritizing them. Muske and Serebrenik give a comprehensive overview of the approaches that have been suggested so far [134].

To visualize data in a structured way, UAV uses treemaps on its package and class level views and an enriched source code view on individual files. Treemaps are a space-filling visualization method that can display large hierarchical collections of quantitative data intuitively [135]. This makes it ideally suited to present the nested structure of a typical JAVA project. UAV uses a modified treemap view to provide an intuitive high-level visualization of which warnings lie where in a project and a seamless switch to a source-level view to track warnings down to individual source code lines.

Tools

Apart from the plethora of individual ASATs available today, tools such as Google's Tricorder [136], Teamscale [137], SonarQube [138], or CoverityScan [139] can collect and display the warnings of multiple ASATs, the first step of UAV. UAV goes further in that it also categorizes the warnings from the multiple tools into one mutual topology, GDC, and visualizes them. Alternatively, UAV displays the ASAT warnings originate from, down to the source code level. Existing tools lack these two capabilities.

2.7.4 Implementation

In this section, we first give an overview of the workflow for a user, then describe UAV's architecture and inner workings, and conclude with a series of technical challenges.

Workflow

Figure 2.8 depicts the typical workflow of UAV. It begins with the user running `maven site` to produce the warning files of the ASATs ①. The user then indicates, in UAV's UI ②, the source folder of the project to analyze. UAV gathers context data on the project and parses the generated ASAT warnings ③. Subsequently, it classifies and groups warnings ④ by applying the GDC ⑤ on them. Next, it writes out the result files for the visualizer ⑥. Finally, UAV opens the user's web browser and runs the visualizer ⑦.

Architecture

Figure 2.9 depicts the two components UAV comprises. The ASATCollector ① gathers and interprets the output generated by running the supported ASATs via Maven. Because of its static and computation-intense nature, we have implemented the ASATCollector in Java 1.8. The ASATVisualizer ②, allows a user-interactive exploration of these warnings transferred from the ASATCollector ③. To emphasize platform independence, speed, and user interaction capabilities, we have implemented the visualizer in JavaScript to run in the user's browser.

The ASATCollector first finds all warnings, along with their specific location in the project, and groups them together. Second, it determines the structure of a project. When one runs UAV, the ASATCollector will open up a JavaFX UI where the user can select the source folder of the project. Once selected, the parsers of the ASATCollector read the warning files generated by `maven site` for Checkstyle, FindBugs and PMD. We use jsoup to parse GDC's ASAT mapping, specifying which ASAT warning to map to which common GDC category. The groupers summarize these warnings according to the read-in GDC. Simultaneously, the ASATCollector gathers information on the structure of the project by looking up all classes within each package, the path to each Java class file and the

2

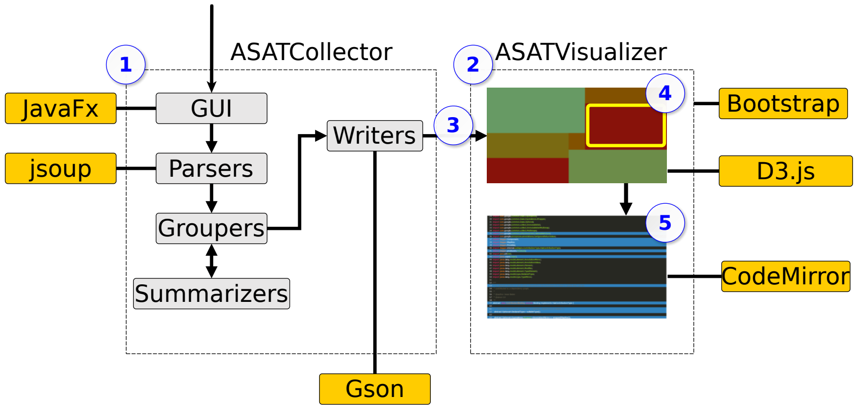


Figure 2.9: Architecture of UAV.

number of lines of code for each class and package. The last step is to write the collected warnings and data to a JavaScript file where it is stored in JSON format and transferred to the ASATVisualizer. The Gson library is used for the creation of JSON objects.

After it creates the output file, UAV opens the user's default browser and shows the visualization. Moreover, users can share its light-weight output file and without having to distribute the visualization code. This also means that multiple users can analyze the produced warnings of the project without having to run the independent ASATs multiple times. The visualization itself is a ready-made template based on the Bootstrap framework using HTML, CSS, and JavaScript. It only requires a JSON output file from the ASATCollector to display its information. In the ASATVisualizer, the treemap in the center of the visualization is implemented using D3.js, a popular JavaScript library for manipulating documents based on data. We have chosen D3.js because of its interactive features and freedom of customization. This enabled us to implement the different filter options of the treemap in pure JavaScript. If the user clicks on a class in the treemap, UAV will seamlessly swap the treemap with the source code viewer. The source code viewer is built using the JavaScript library CodeMirror; we modified the syntax highlighting to show the warnings at the source code level with color-coding.

ASATVisualizer User Interface

In this section, we describe the two main UI components of UAV: Its treemap high-level package view depicted in Figure 2.11 and its source code-level view in Figure 2.10.

UAV's visualization provides users with a large treemap showing the structure of the project (① in Figure 2.11). The treemap can be navigated through by clicking on the desired block. Currently, the package 'dagger.internal.codegen' is highlighted ②. Next to the mouse cursor, UAV displays a pop-up with descriptive statistics about the highlighted package, such as its number of warnings per ASAT ③. The user could click on this package to zoom in on it. In the menu on the left ④, users can select which ASATs to include in the visualization. They can adjust which metric the color of the classes are based on:

- 'Normal' shows the amount of warnings relative to other classes.

Figure 2.10: Code-level view of UAV with Checkstyle and PMD warnings.

- ‘ASAT’ shows the distribution of which ASAT the warnings originate from.
- ‘Category’ shows the distribution of warnings according to which of the GDC categories (functional defects, maintainability defects, or other) they belong to.

When in ‘Normal’ color scale, users can also choose to base the intensity of the colors on the relative amount of warnings in each class or on an absolute scale (where pure green means no warnings and pure red means one warning per line). In the GDC panel on the right side ⑤, the user can see the warning categories and toggle them on or off.

The user can navigate down from package level into class level view, and finally view a single class on code level, shown in Figure 2.10. UAV color-codes each line with a warning, see line 16. According to the setting of ④ in Figure 2.11, the color can indicate which ASAT the warnings originate from or which category they belong to. In lines with multiple warnings, colors alternate, see line 5. It contains a warning about code structure, namely the import ‘java.io.IOException’ is not used. Both PMD and Checkstyle have reported this warning.

Challenges

We have encountered three major challenges during the development of UAV. Our first challenge was to find a way for UAV to run the ASATs. The initial solution was to use processbuilder from Java; it is possible to run commands via processbuilder to execute the ASATs. However, this solution required an executable of each ASAT, which restricts our users to one version of an ASAT and makes it difficult to update. Our alternate solution is to use Maven to produce the output files of the ASATs. For this solution there is no need to package third-party executables of ASATs together with UAV.

The solution for the first challenge, however, is a cause of the remaining two challenges: we had to find a way for UAV to run Maven and to gather all the output files of the ASATs.

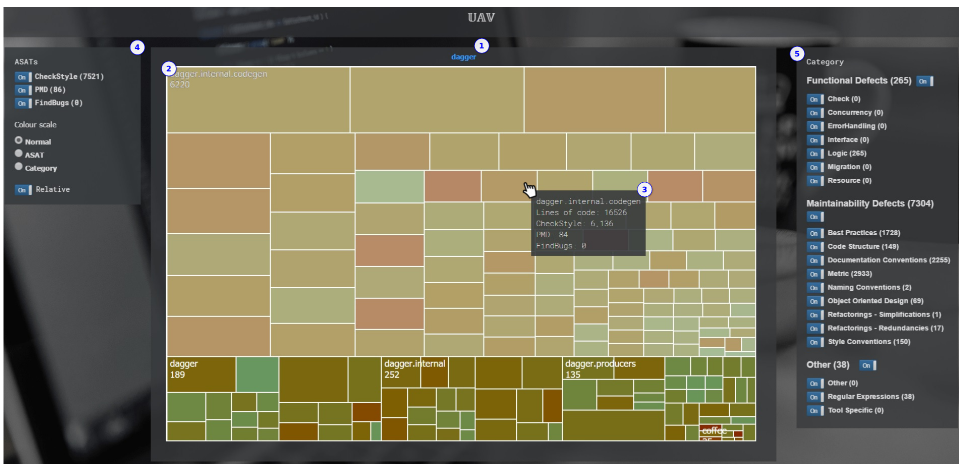


Figure 2.11: High-level package view of UAV on the Dagger project.

As Maven can be run as a standalone application, installed in the system, or incorporated in an integrated development environment (IDE), it is difficult to determine and support all three possible installation scenarios through UAV. Instead, we have therefore decided to let the user run Maven on their project before they use UAV.

Finding the warning files of ASATs is a straight-forward task as long as a project only contains one Maven configuration file. However, in many larger projects, each package has its own pom configuration file, which produces its own ASAT outputs. Hence, before UAV can work on these, it must unify them into a single warning file per ASAT.

2.7.5 Evaluation

In this section we report on initial evaluations of UAV on three real-world systems and a usability study with ten CS students.

Project Case Study

To evaluate whether UAV can be used on larger real-world projects, we tested it on two popular Java projects from GitHub, google/dagger (5,292 stars)[4] and apache/curator (486 stars),[5] and on itself (the Java part of UAV, ASATCollector in Figure 2.9). For each project, we ran the tool ten times and calculated the average run time. We measured the run time from when the tool starts to gather all information of the user's project to the point where the analyzer writes the output files for the ASATVisualizer. Table 2.7 shows descriptive statistics and the run time of our tool on each project.

An interesting result from comparing the tree projects was that the amount of warnings per tool depended on the project, their specific ASAT configuration. For example if many Checkstyle rules are removed or FindBugs is set to a lower rigidity, then the amount of warnings is visibly reduced in UAV. We could compare and observe the effect of mod-

[4] `https://github.com/google/dagger`

[5] `https://github.com/apache/curator`

Table 2.7: Descriptive statistics of UAV on three example projects.

Name	#LOC	#Checkstyle warnings	#PMD warnings	#FindBugs warnings	Run time
google/dagger	59,864	7,521	86	0	73s
Netflix/curator	122,094	16,691	53	0	156s
UAV	4,796	5	20	14	1s

ifying the project's configurations via UAV's "absolute" color scheme (see Section 2.7.4). Thus, UAV also provides insights in the development stage of software.

User Study

We invited ten second year computer science students and later a visualization expert (both with no prior knowledge of UAV) to participate in our usability testing. We placed them in front of a computer with UAV, accompanied by a list of questions, and a short explanation of the purpose of the tool. The testers could interact with the tool while answering questions related to its use. Questions like "Which package has the most warnings?" and "How many warnings in the project are about Code Structure?" helped us assess how intuitive to use UAV was by measuring how many students delivered a correct answer. The last question was an open question where the testers were asked for further feedback. We replicate the list of all questions and the in-depth results of the usability evaluation in an online appendix [140].

Our results indicate that most testers understood the goal of the tool. At least 70% of respondents answered each question correctly. Based on incorrect answers and the feedback given in the evaluation, we could improve the tool in several ways. One such improvement is the backwards navigation bar. One of the testers said: "The back button on the top looks like you can go back to a specific folder instead of the previous folder." This feature was initially designed to allow users to go one level up in the visualization of their project. After discussions within the team, we replaced the navigation feature with the current path to the file which the user is viewing. Moreover, we made each component of the path itself clickable. We could implement several more improvements in the UI and UAV's usability. Later feedback from the visualization expert showed us that this made the navigation of the tool more intuitive [140].

2.7.6 Development Roadmap

In this section, we describe possible improvements and extensions of UAV for future work.

Due to compatibility issues with the treemap visualization and the gradient color representation of D3.js, Chrome and Safari are the only supported browsers at this time. We plan to resolve the cosmetic problems with Firefox.

UAV's visualization of nested packages could be improved. It currently does not show the nested relationship of sub-packages, but rather includes them on the top-level of the treemap. Implementing this feature would allow UAV to handle more hierarchically complex projects.

The current UAV prototype supports three Java ASATs. A natural improvement would be adding more ASATs to broaden the selection of tools that can be compared by including tools such as Google's Error Prone. The ASATCollector facilitates adding new ASATs

thanks to its modular structure. We would only need to change the UI of the ASATVisualizer to handle the visualization of additional tools. Supporting more tools and programming languages would also lift UAV's status of a prototype.

A promising avenue of future work would be the integration of UAV with GitHub and Travis CI, a cloud service that automatically builds GitHub projects. Similar to CodeClimate, a new commit on GitHub could trigger the execution of Maven on Travis CI, store the ASAT warnings as build artifacts, and UAV in the cloud would collect these artifacts and generate a JSON file for the visualization. The existing visualization implementation of UAV lends itself toward such hosting in the cloud, since it is based on a web-stack and would only require the relatively light-weight visualization file locally.

2.8 Future Work & Conclusions

In this chapter, we have performed an investigation into how a large set of OSS projects use static analysis. Our findings show that, 60% of the most popular and (therefore arguably) most advanced projects make use of ASATs. Projects which use ASATs typically do not embed them in their workflow and use them only sporadically. Our results seem to suggest that dynamically-typed languages benefit from or require more ASAT support than static languages. Future research could broaden the group of languages for this analysis to assert and further investigate this finding.

Our analysis into the usage of ASATs through their configuration files has shown that the default configurations of most tools are a good fit to the majority of projects. Only two tools contained default checks that developers regularly disagreed with. In line with the picture of a light use of ASATs are our results on the evolution of their configuration files: There typically is no evolution. Most ASAT configurations, after an initial period of change of one week, remain unchanged in project repositories.

Our findings seem to suggest that OSS developers need to be made aware of the benefits of using ASATs, and how easy an integration into their fixed workflow or even continuous integration process can be. On the other hand, developers might be skeptical of the practical usefulness of ASATs due to a possible overload with irrelevant warnings.

In addition to the purely empirical analysis, this chapter also described UAV, a tool that provides an intuitive way to compare multiple ASATs. UAV makes the following key contributions:

- A novel structured, interactive visualization that allows for comparison between multiple ASATs.
- Configuration options to switch the visualization between the amount of warnings per ASAT, package, class and GDC defect type.
- A basic framework that can be expanded to include more ASATs and comparison methods as well as additional features.
- A clear overview of warnings from different ASATs in large real-world software projects.

In our first evaluation, our UAV prototype has demonstrated its capability of visualizing warnings by clearly representing multiple Java projects of different project sizes and

ASAT warning densities. Users of our tool have a more coherent view of the types and locations of warnings as indicated by different ASATs. Our vision is that one day, anyone who uses code analysis can input their preferences, and UAV will combine different ASATs to output a result that best suits their needs.

In conclusion, this chapter also contributes the GDC and practical guidelines for users and creators of ASATs. Possible benefiters are:

Researchers, who can replicate our study and use the classification for further studies on ASATs. The GDC might be especially useful for studies on the intersection of ASATs with code review [141].

Practitioners, who could assess the strengths and weaknesses of ASATs by inspecting the distribution profile of the number of supported checks in each category. For example, FindBugs emphasizes functional checks.

Tool Creators of FindBugs and RuboCop, who may want to re-assess the defaults for their rules in two GDC categories. Developers seem to accept the remainder of the defaults.

Dashboard Creators of tools, such as Teamscale [142] and SonarQube [138], who could rank, compare, filter, prioritize, and possibly remove duplicates when they assemble warnings from multiple ASATs in one location. Our purpose-built tool UAV can support them in this task.

3

3 The Last Line Effect Explained

Micro-clones are tiny duplicated pieces of code; they typically comprise only few statements or lines. In this chapter, we study the "Last Line Effect," the phenomenon that the last line or statement in a micro-clone is much more likely to contain an error than the previous lines or statements. We do this by analyzing 219 open source projects, reporting on 263 faulty micro-clones, and interviewing six authors of real-world faulty micro-clones. In an interdisciplinary collaboration, we examine the underlying psychological mechanisms for the presence of these relatively trivial errors. Based on the interviews and further technical analyses, we suggest that so-called "action slips" play a pivotal role for the existence of the last line effect: Developers' attention shifts away at the end of a micro-clone creation task due to noise and the routine nature of the task. Moreover, all micro-clones whose origin we could determine were introduced in unusually large commits. Practitioners benefit from this knowledge twofold: 1) They can spot situations in which they are likely to introduce a faulty micro-clone and 2) they can use PVS-Studio, our automated micro-clone detector, to help find erroneous micro-clones.

This chapter has been published as *M. Beller, A. Zaidman, A. Karpov, and R.A. Zwaan. The Last Line Effect Explained, EMSE'17* [62], an extension of *M. Beller, A. Zaidman, A. Karpov. The Last Line Effect, ICPC'15 (ERA)* [63].

Software developers oft need to repeat one particular line of code several times in succession with only small alterations, like in this example from TrinityCore:[1]

Example 3.1: TrinityCore

```
x += other.x;
y += other.y;
z += other.y;
```

The 3D-coordinates of the `other` object are added onto the member variables representing the coordinates `x, y, z`. However, the last line in this block of three similar lines contains an error, as it adds the `y` coordinate onto the `z` coordinate. Instead, the last line should be

```
z += other.z;
```

3

Another example from the popular web browser Chromium[2] shows that this effect also occurs in similar statements within one single line:

Example 3.2: Chromium

```
std::string host = ...;
std::string port_str = ...;
if (host != buzz::STR_EMPTY && host != buzz::STR_EMPTY)
```

Instead of comparing twice that `host` does not equate the empty string, in the last position, `port_str` should have been compared:

```
if (host != buzz::STR_EMPTY && port_str != buzz::STR_EMPTY)
```

Lines 1–3 from Example 3.1 are similar to each other, as are the statements in the if clause in line 3 from Example 3.2. We call such an extremely short block of almost identically looking repeated lines or statements a *micro-clone*. Through our experience as software engineers and software quality consultants, we had the intuition that *the last line or statement in a micro-clone is much more likely to contain an error than the previous lines or statements*. The aim of this chapter is to verify whether our intuition is indeed true, leading to two research questions:

RQ II.1 Is the last line in a multi-line micro-clone more likely to contain an error?

RQ II.2 Is the last statement in a single-line micro-clone more likely to contain an error?

As recurring micro-clones are common to most programming languages, the presence of a last line effect can impact almost every programmer. If we can prove that the last of a series of similar statements is more likely to be faulty, code authors and reviewers alike will know which code segments to give extra attention to. This can increase software quality by reducing the amount of errors in a program.

One natural way to come up with code as in Examples 3.1 and 3.2 is to copy-and-paste it. By closely examining the origin of micro-clone instances, we come to the conclusion that developers employ a variety of different mechanical patterns to create micro-clones, most

[1] TrinityCore is a popular open-source framework for the creation of Massively Multiplayer Online Games (MMOGs), `www.trinitycore.org`.

[2] Chromium is the open-source part of Google Chrome, `www.chromium.org`

important among which is copy-and-pasting on a line-per-line basis. Copy-and-pasting and cloning are some of the most widely used idioms in the development of software [143]. They are easy and fast to do, hence cheap, and the copied code is already known to work. Though often considered harmful [144], sometimes (micro-)cloning is in fact the only way to achieve a certain program behavior, like in the examples above. A number of clone detection tools have been developed to find and possibly remove code clones [145, 146]. While these automated clone detection tools have produced very strong results down to the method level, they are ill-suited for recognizing micro-clones in practice because of an abundance of false-positives.

When we posted a popular science blog entry[3] about the last line effect, it was picked up quickly and excitedly in Internet fora.[4] Many programmers agreed to our observation, often assuming a psychological reason to cause the effect. This leads to our last research question:

RQ II.3 What are the reasons for the existence of faulty micro-clones and the last line effect in particular?

Through interviews, deeper technical analyses and interdisciplinary work with a psychologist, we research whether and which psychological aspects could play a role in the last line effect. We first collect phenomena from the well-established area of cognitive psychology and then research if they explain the last line effect in micro-clones.

By building upon our initial investigation of the last line effect [63], we make the following contributions:

- We define and introduce the term *micro-clone*.
- We introduce techniques for the detection of faulty micro-clones implemented in the automated static analysis tool (ASAT, [60]) PVS-Studio, which cannot be found with traditional clone detection.
- We manually investigate the error proneness of 263 micro-clones in 219 well-known open-source systems (OSS), based on a total of 1,891 warnings.
- We provide an initial analysis of the underlying psychological mechanisms behind the existence of the last line effect.
- We lead six interviews with authors of micro-clones in real-world systems.
- We conduct a repository analysis on four well-known OSS projects based on the results of the interviews, investigating abnormally large commit sizes.

Our findings show that in micro-clones similar to Examples 3.1 and 3.2, the last line or statement is significantly more likely to contain an error than any other preceding line or statement. Rather than technical complexity of the micro-clone, psychological reasons seem to be the dominant factor for the existence of these faulty micro-clones, mostly related to working memory overload of programmers. An initial investigation with

[3] www.viva64.com/en/b/0260

[4] www.reddit.com/r/programming/comments/270orx/the_last_line_effect

five projects reveals that all micro-clones were introduced in abnormally large commits at often unusual work hours. This knowledge and our ASAT PVS-Studio can support human programmers in reducing the amount of simple last line-type of errors they commit by automating the detection of such faulty pieces of code.

3.1 Study Setup

Our study consists of two empirical studies C_1 and C_2. In this section, we describe how we set-up the two empirical studies on micro-clones and which study objects we selected.

3.1.1 Study Design C_1: Spread and Prevalence of the Last Line Effect within Micro-Clones

Study C_1 examines how wide-spread the last line effect is within micro-clones. We statistically examine the existence of the last line effect within micro-clones in five easily replicable steps. Moreover, in an effort to shed light on how developers create them, we added an analysis on the origin and destination of micro-clone instances.

1. Run the static analysis checker PVS-Studio on our study objects, with all checks enabled. PVS-Studio is a commercial static analysis tool developed by the Russia-based company “OOO Program Verification Systems” and incorporates dozens of static analyses from detecting clones to recognizing anti-patterns of using specific library functions in C. For replication purposes, a free trial of PVS-Studio is publicly available.[5]

2. Inspect the corpus of warnings from PVS-Studio and remove false-positives and warnings not related to micro-cloning.

3. For each faulty micro-clone, count the total number of lines (RQ II.1) or statements (RQ II.2) and denote which lines or statements are faulty. If possible, infer the likely origin and destination of the micro-clone (for example, in Example 3.6, the origin would be line 2 and the destination line 3).

4. Naively, we assume each line has the same likelihood $1/n$ of containing an error (H_0), independent of its position in an n-line long faulty micro-clone. For example, lines 1 and 2 in a 2-liner clone each have a 0.5 probability of containing an error. However, if we can show that the error distribution per line from step (3) significantly differs from such a uniform distribution on a $\sigma = 0.05$ significance level, we reject H_0 and assume a non-uniform error distribution. For each micro-clone length n, we use Pearson's χ^2 test with $n - 1$ degrees of freedom to compare our empirical distribution's goodness-of-fit to a $1/n$-equipartition.

5. If the test in step 4 finds a significant difference between the two distributions, we calculate the odds ratio between them as an intuitive measure of how strong the effect size of the last line effect is [147].

[5] www.viva64.com/en/pvs-studio-download

3.1.2 Study Design C_2: Analyzing Reasons Behind the Existence of the Last Line Effect

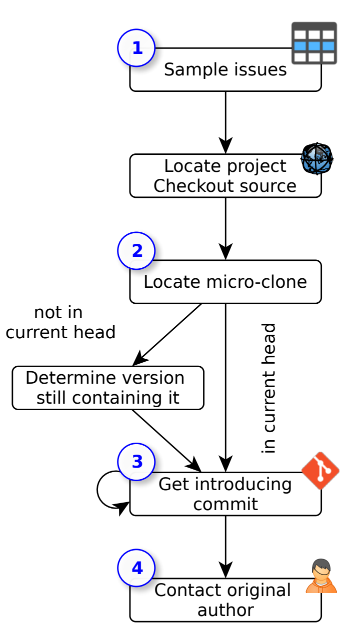


Figure 3.1: Study Design of C_2.

Having established the existence of the last line effect in C_1, we want to gain insight into the reasons why it exists (RQ II.3). To this aim, we build an initial theory based on related work in the domain of cognitive psychology together with Rolf Zwaan, professor of cognitive psychology. To obtain anecdotal evidence on micro-clones through interviewing developers and to further corroborate these cognitive explanations, in study C_2, we interview developers that authored last line effect instances. We specifically interviewed developers who authored micro-clones that we found in C_1. The emerging observations will aid us in creating an initial psychological explanation. By only contacting developers who we knew had created a micro-clone, we make our interviews (1) more focused on a concrete instance of the phenomenon that our interviewees could personally relate to and (2) more relevant by approaching an audience that we could prove had authored a faulty micro-clone in the past.

Figure 3.1 depicts our general study design. It centers around finding and contacting the original author of a micro-clone which in many cases is not present in the project's latest checkout anymore. The design comprises four primary steps:

1. We randomly sample projects and micro-clones to investigate from these projects, since carrying out C_2 is a tedious manual process that involves contacting and interviewing developers. Assuming a standard response rate for cold calling surveys of 30%, the resulting three interviews would likely give us sufficient information to guide the creation of our initial psycho-cognitive explanation. For each micro-clone, we have to familiarize us with the project's development guidelines and check out their repository.

2. Next, we locate the micro-clone in the source tree of the project. Since many projects fixed our observations in the meantime and the clone is then not present in the current head, this step requires different search strategies: we start by checking out the repository at the date of our inspection for C_1. If this fails, for example because the code around the micro-clone was refactored (or the history force-overwritten), we try to track down the commit that removed the micro-clone by searching the project's issue tracker. If all else fails, we resort to a full text search (via ag) in every commit of the project's history.

3. Once we tracked down the original micro-clone, we follow its history, using `git`

blame, to ensure we receive both refactorings that were applied to it as well as its true original author.

4. In the last step, we use `git blame -e` to obtain the developers' email addresses to contact them. In an attempt to maximize the response rate, we also perform a web search to acquire additional personal information about the developers and verify the timeliness of the contact email addresses. We also made clear we will not disclose their identity to incentivize honest answers. We then send short personalized emails containing the micro-clone they authored, how it was later modified or fixed, the context of the bug, why we do the investigation, and a set of questions on the micro-clone at hand.

3.1.3 Study Objects

To ensure the replicability and feasibility of our study, we focused on well-known open-source systems. Among the 219 OSS we studied in C_1, we found instances of defective micro-clones in such renowned projects as the music editing software Audacity (1 finding), the web browsers Chromium (9) and Firefox (9), the XML library libxml (1), the databases MySQL (1) and MongoDB (1), the C compiler clang (14), the ego-shooters Quake III (3) and Unreal 4 (25), the rendering software Blender (4), the 3D visualization toolkit VTK (8), the network protocols Samba (4) and OpenSSL (2), the video editor VirtualDub (3), and the programming language PHP (1). For C_2, we sampled 10 micro-clones from the projects Chromium, libjingle, Mesa 3D, and LibreOffice.

3.1.4 How to Replicate This Study

To foster replication of this study, we have made the complete data set and all analyses available as a replication package.[6] The package includes all un-filtered warnings from PVS-Studio, separated into the older data used for our ICPC paper [63] (`findings_old/`) and the newer data added for this chapter (`findings_new/`). Moreover, it contains the analyzed and categorized micro-clones (`analyzed_data.csv`) together with an evaluation spreadsheet (`evaluation.ods`) and the results from the repository analysis of C_1 and C_2. We also provide the R scripts to replicate the results and graphs in this chapter. Finally, we share a draft of the questions we sent to developers.

3.2 Methods

In this section, we outline traditional clone detection, why it is ill-suited for the recognition of micro-clones, and how we circumvented this problem with our tail-made static checks, our origin inference of micro-clone instances and commit size analysis.

3.2.1 Inaptness of Current Clone Detectors

As Examples 3.1 and 3.2 demonstrate, the code blocks that we study in this chapter are either textually identical or contain "syntactically identical cop[ies]; only variable, type, or function identifiers have [...] changed." [148] This makes them *extremely short type-1 or type-2 clones* (usually shorter than 5 lines or statements), which we refer to as *micro-clones*.

[6] `http://dx.doi.org/10.6084/m9.figshare.1313697.`

Traditional code clone detection works with a token-, line-, abstract syntax tree (AST), or graph-based comparison [148]. However, to reduce the number of false-positives, all approaches are in need of specifying a minimal code clone length for their unit of measurement (be it tokens, statements, lines or AST nodes) when applied in practice. This minimal clone length is usually in the range of 5–10 units [145, 149], which makes it too long to detect our micro-clones of length 2 to 5 units.

Consider Example 3.1, in which all lines 1–3 together form the micro-clone class. There are three micro-clones of this class, since each line 1–3 represents a single instance. Every micro-clone instance in Example 3.1 consists of a variable, an assignment operator and the assigning object and its member variable, so its length in abstracted units is four.[7]

3.2.2 How to Find Faulty Micro-Clones Instead

As clone detection is not able to reliably detect micro-clones in practice, we devised a different strategy to find them. Our research questions aim not at finding all possible micro-clones, but only the ones which are faulty. With this additional constraint, we could design and implement a handful of powerful analyses based on simple textual identity. These are able to find faulty code that is very likely the result of a micro-clone. Table 3.1 lists and describes the twelve analyses that found micro-clones in our study.[8] The last column summarizes the within- and multi-line code clones onto the number of all warnings found for this error code. For example, the analysis V501 simply evaluates whether there are identical expressions next to certain logical operators. If so, these are at best redundant and therefore cause a maintenance problem, or at worst, represent an actual bug in the system. Other analyses are not as specific toward micro-cloning as V501. We inspected all 526 warnings manually and only included the 272 containing an actual micro-clone in our study. Table 3.1 also shows that 78% of our micro-clones stem from one analysis only (V501), which has a very low false-positive rate of 97%. Other analyses have a higher likelihood of not only being triggered by micro-clones.

3.2.3 Inferring the Origin of an Erroneous Micro-Clone Instance

To be able to make qualified statements about why a last line effect might exist in RQ II.3, we additionally identify, for each micro-clone class, the copied erroneous clone instance and the instance it likely originated from. While this a-posteriori analysis cannot provide us with absolute certainty that the clones were created in this way, we have convincing evidence that at least some developers mechanically create micro-clones this way (see RQ II.3). As in Example 3.1, in the majority of cases, it is most often immediately clear which is the influencing and which the influenced unit in a micro-clone: The erroneous line 3 contains left-over fragments from line 2, implying an influence from 2 (origin) to 3 (destination). Most micro-clones exhibit a similar natural order that determines origin and destination, either lexicographically like `x`, `y`, `z` in Example 3.1 or cardinally:

[7]Some clone detectors would count the assigning object and the reference to the member variables as one unit. Following this definition, the length in units would be three.
[8]For a more detailed description of the analyses, refer to `http://viva64.com/en/d/0368/`.

Table 3.1: Error Types from PVS-Studio and Their Distribution in our 219 OSS systems.

PVS Error Code	Description	#within line clones	#multi line clones	Σ#/All
V501	There are identical sub-expressions to the left and to the right of the `foo` operator.	104	108	212/217
V517	The use of **if** (A){...} **else if** (A){...} pattern was detected. There is a probability of logical error presence.	0	8	8/58
V519	The `x` variable is assigned values twice successively. Perhaps this is a mistake.	0	23	23/117
V523	The `then` statement is equivalent to the **else** statement.	0	5	5/47
V524	It is odd that the body of `Foo_1` function is fully equivalent to the body of `Foo_2`.	0	3	3/13
V525	The code containing the collection of similar blocks. Check items `X, Y, Z, ...` in lines N1, N2, N3, ...	1	1	2/11
V537	Consider reviewing the correctness of `X` item's usage.	0	8	8/10
V570	The variable is assigned to itself.	1	1	2/17
V571	Recurring check. This condition was already verified in previous line.	0	2	2/17
V581	The conditional expressions of the **if** operators situated alongside each other are identical.	0	2	2/13
V583	The `?:` operator, regardless of its conditional expression, always returns one and the same value.	0	1	1/7
V656	Variables are initialized through the call to the same function. It's probably an error or un-optimized code.	0	4	4/8
Σ		106	166	272/535

Example 3.3: CMAKE

```
p[0] = 0xfc | ((wc >> 30) & 0x01);
p[1] = 0x80 | ((wc >> 24) & 0x3f);
p[1] = 0x80 | ((wc >> 18) & 0x3f);
p[2] = 0x80 | ((wc >> 12) & 0x3f);
```

Even in cases where there is no explicit natural order as in Examples 3.1 and 3.3, the code context often motivates an implicit order, like in Example 3.2: It would be against the order of their previous definitions to put `port_str` in the first place and `host` in the second place in line 3. Hence, we assume that the first instance of the micro-clone `host != buzz::STR_EMPTY` is the influencing origin and the second instance is the destination.

The general problem when reasoning about the origin and destination of micro-clones in our data set is 1) the possible variable clone length and 2) the expected relatively fewer micro-clones of length greater than 4. In order to be able to generalize over different micro-clone lengths nonetheless, we calculate, for each micro-clone i, $\delta_i = \text{line}_i(Destination) - \text{line}_i(Origin)$, resulting in the proximity distribution $\Delta_{Dest-Orig}$.

A value of 1 indicates an inference from the immediately preceding unit, as in Example 3.3. A value of 0 denotes that the error occurred within the same micro-clone instance. A value of -1 denotes a swapped pair of clones, in which the second influenced the first:

Example 3.4: UnrealEngine4

```
1 return cy().isRelative()
2     || cy().isRelative()
3     || r().isRelative()
4     || fx().isRelative()
5     || fy().isRelative();
```

Here, we would have expected `cx().isRelative` in line 1, instead of `cy().isRelative`, which seems to be influenced by the second line. Natural order, as well as lines 3 and 4 suggest that the micro-clone start with `return cx().isRelative()` in line 1 instead.

Hence, adding up the number of values where $\Delta_{Dest-Orig} = 1$ or $\Delta_{Dest-Orig} = -1$ gives us the number of clones that are direct neighbors to each other, either on the same line or the next line, irrespective of the total length of the clone.

3.2.4 Putting Commit Sizes in Perspective

To calculate and visualize how each micro-clone inducing commit relates to the remaining distribution of commit sizes, we first calculate the churn for each commit in the repository. We do this by instrumenting `git log` to build a sequenced graph of all commits (excluding merges) in the repository, extracting the number of added and deleted lines in each commit, summing them up as the modified lines and outputting this churn integer for each commit. We then compare the churn of the micro-clone inducing commits to the overall distribution, and specifically to its median. Although our sample size of ten is too small for statistical testing, this way, we can make substantiated statements about a possible size difference between commits. We use the median (and not the average mean, for example) as our distributions are non-normal, it is a single real value and we compare other, singular observations to it.

3.3 Results

In this section, we deepen our understanding of faulty micro-clones by example and statistical evaluation.

3.3.1 Overview Description of Results

Table 3.2 presents basic descriptive statistics of our results for C_2. We ran the complete suite of all PVS-Studio analyses on our 219 OSS from mid-2011 to July 2015. Andrey

Table 3.2: Descriptive statistics of study results.

	... with all findings	**... with faulty micro-clones**	**... with last line or statement bug (rel. to all, rel. to faulty)**
Analysis time	June 2011 to July 2015		
Analysis software	PVS-Studio versions 4.00 to 5.27		
# of analyses	162	12 (7%)	12 (7%, 100%)
# of projects	219	106 (49%)	97 (45%, 92%)
# of warnings	1,891	272 (14%)	228 (12%, 84%)
# of unique clones	–	263 (–)	228 (–, 87%)

Table 3.3: Summarized study results.

	multi-line micro-clone	one-line micro-clone
#errors in last line or statement	*117 (74%)*	*95 (90%)*
#errors not in last line or statement	41 (26%)	10 (10%)
effect size (odds ratio)	2.9	9.5
Σ	158 (100%)	105 (100%)
ΣΣ	263	

Karpov, a software consultant by profession, gradually analyzed the different systems throughout this period, using the latest then-available version of PVS-Studio. He sorted-out false positives, so that 1,891 potentially interesting warnings with 162 different error codes remained. We then manually investigated all 1,891 warnings and found that 272 warnings with twelve distinct error codes were related to micro-cloning. Nine micro-clones contained two such warnings, so that we ended up with 263 micro-clones. The statistics at the project level reveal that our analyses could identify faulty micro-clones in half of the investigated projects. Almost all of these (92%) contained at least one instance of the last line or statement effect.

Table 3.3 presents a high-level result summary of locating errors in 263 micro-clones. In total, we see that 74% of multi-line micro-clones have a last line error and 90% of one-liner micro-clones in the last statement.

3.3.2 In-Depth Investigation of Findings

To convey a better intuitive understanding of the analyses with which we identify faulty micro-clones, in the following, we select some of the 263 PVS-Studio-generated micro-clone warnings as representative examples for the most frequently occurring error codes from Table 3.1.

V501 – Identical Sub-Expressions

As Table 3.1 shows, the majority of micro-clone warnings are of type V501. A prime example for a V501-type warning comes from Chromium:

Example 3.5: Chromium

```
1 return !profile.GetFieldText(AutofillType(NAME_FIRST)).empty() ||
    !profile.GetFieldText(AutofillType(NAME_MIDDLE)).empty()||
    !profile.GetFieldText(AutofillType(NAME_MIDDLE)).empty();
```

In this one-liner micro-clone the second and third cloned statement are lexicographically identical but connected with the logical OR-operator (`||`), thus representing a tautology. Instead, the Boolean expression misses to take into account the surname (`NAME_LAST`), an example of the last statement effect in this tricolon.

V517 – Identical if-Conditions

Error code V517 pertains to having identical entry-conditions for two branches of `if`-statements.

Example 3.6: LINUX-3.18.1

```
if (slot == 0)
{
    ...
}
else if (slot == 1)
{
    ...
}
else if (slot == 0)
{
    ...
}
```


The body of the `else if` condition following the third micro-clone on line 9 is dead code, as it can never be reached. If `slot` was indeed zero, it would already enter the first `if` condition's body.

V519 – Identical Assignment to Variable

Setting the value of a variable twice in succession is typically either unnecessary (and therefore a maintenance problem because it makes the code harder to understand as the first assignment is not effective), or outright erroneous because the right-hand side of the assignment should have been different. In the following V519 example from MTASA, `m_ucRed` is assigned twice, but the developers forgot to set `m_ucBlue`.

Example 3.7: MTASA

```
        m_ucRed = ucRed; m_ucGreen = ucGreen; m_ucRed = ucRed;
```

The detection of V519-style warnings works well for most "regular" software, but is to be taken with caution when analyzing firmware or other hardware-near code, as Example 3.8 demonstrates:

Example 3.8: LINUX-3.18.10

```
f->fmt.vbi.samples_per_line = 1600;
f->fmt.vbi.samples_per_line = 1440;
```

The second line sets the value of the variable `f->fmt.vbi.samples_per_line` again, even though it has just been set in line 1. Since no other method calls have been made in the further control flow of this method, the assignment in line 1 seems to have no effect. However, as the assignment is active for at least one CPU cycle, there might be threads that read its value in the meantime (for example, watchdogs on the buffer state) or there might be other intended side-effects. To be on the conservative side, we compiled the code with release settings and if the compiler optimized the first assignment away, we were sure it was indeed an error.

V523 – Equivalent Behavior Despite Branching

When we find a micro-clone for different branches of `if`-conditions, these could be simplified by collapsing them into one block, for example in HAIKU:

Example 3.9: HAIKU

```
if (flags & ATTR_COMPRESSION_MASK) {
  hdr_size = 72;
```

```
3      // FIXME: This compression stuff is all wrong. .... /
4      // now. (AIA) /
5      if (val_len)
6        mpa_size = 0; // get_size_for_compressed...; /
7      else
8        mpa_size = 0;
9      ...
10  }
```

It is, however, more likely that `mpa_size` should have been set to a different value in the `else`-branch. The code context of this micro-clone seems highly suspicious, as it mentions in line 3 that "[t]his compression stuff is all wrong," and the detected erroneous micro-clone fits to this comment.

V524 – Equivalent Function Bodies

Two cloned functions with the same content are highly suspicious. In our Example 3.10, line 5 should call `PerPtrBottomUp.clear()`. This also serves as one rare example of a two-instance micro-clone where the origin succeeds the target ($\delta_{E_{10}} = -1$).

Example 3.10: CLANG

```
1  MapTy PerPtrTopDown;
2  MapTy PerPtrBottomUp;
3
4  void clearBottomUpPointers() {
5    PerPtrTopDown.clear();
6  }
7
8  void clearTopDownPointers() {
9    PerPtrTopDown.clear();
10 }
```

V537 – Suspicious Use of Variable or Statement

An illustrative example for a V537 finding comes from QUAKE III, where PVS-STUDIO alerts us to review the use of `rectf.X`:

Example 3.11: QUAKE III

```
1  rect->X = roundr(rectf.X);
2  rect->Y = roundr(rectf.X);
```

The rectangle's y-coordinate is falsely assigned the rounded value of `rectf.X` in the second (i.e., last) line of this micro-clone.

V656 – Two Variables Bear Identical Value

V656 checks for different variables that have the same initializing function. As a result, we need to check warnings of type V656 carefully, as they bear a high potential for false-positives. One example for a false-positive is that the two variables are supposed to start with the same value, and are then treated differently in the downstream control-flow. All V656-related micro-clones in our sample stem from LIBREOFFICE.

Example 3.12: LIBREOFFICE

```
1  maSelection.Min() = aSelection.Min();
2  maSelection.Max() = aSelection.Min();
```

Here, `maSelection.Max()` is assigned not the maximum value of `aSelection`, but its minimum, clearly representing an error.

Counterexample

As we have already seen in Example 3.12, not for all instances of an erroneous micro-clone does the problem lie in the last line or statement. Take this rare counterexample from Chromium, which we counted towards the 12 instances of an error in line 2 of a three-liner micro-clone (see Table 3.4):

Example 3.13: Chromium

```
1 if (std::abs(data_[M01] - data_[M10]) > epsilon ||
2     std::abs(data_[M02] - data_[M02]) > epsilon ||
3     std::abs(data_[M12] - data_[M21]) > epsilon)
```

In line 2, the engineers deducted `data_[M02]` from itself. However, they meant to write:

```
2     std::abs(data_[M02] - data_[M20]) > epsilon ||
```

3.3.3 Statistical Evaluation

Table 3.4 shows the error-per-line distribution of our 158 micro-clones consisting of several lines, and Table 3.5 that of our 105 micro-clones within one single line. Cells with gray background are non-sensible. For example, in a micro-clone of 2 lines length, no error can occur in line 3. The yellow diagonal highlights errors in the last line or statement.

For each column in Tables 3.4 and 3.5, we performed a Pearson's χ^2 test on a $p = 0.05$ significance level to see whether the individual distributions are non-uniform. The resulting p-values, reported in the last row, are only meaningful for micro-clone lengths with enough empirical observations, which are columns 2–6 in Table 3.4 and columns 2–4 in Table 3.5.

Table 3.4: Error distribution for micro-clones with $\geqslant 2$ lines.

#errors in line	#total lines: 1	2	3	4	5	6	7	8	9	>9
1		8	0	0	1	0	0	0	0	0
2		*66*	12	3	1	0	0	0	0	0
3			*22*	4	0	1	1	0	0	0
4				*15*	0	0	0	0	0	0
5					*6*	1	0	1	0	1
6						*3*	0	0	0	1
7							*1*	0	0	1
8								*0*	1	0
9									*2*	4
>9										*2*
Σ	0	74	34	22	8	5	2	1	3	9
ΣΣ	158									
p		10^{-106}	10^{-27}	10^{-15}	10^{-5}	0.0487	0.534	0.437	0.135	

For RQ II.1 and RQ II.2, we received significant p-values for micro-clones consisting of 2, 3, 4, 5 or 6 lines and for micro-clones consisting of 2, 3, or 4 statements ($p < 0.05$). This means that we can reject the null hypothesis that errors are uniformly distributed across statements or lines. Instead, the distribution is significantly skewed towards the last line or statement. We would expect similar findings for longer micro-clones, but these were

Table 3.5: Error Distribution for Micro-Clones within One Line.

#errors in statement	1	2	3	4	5	> 5
1		1	0	0	0	0
2		*71*	4	2	0	0
3			*18*	1	0	0
4				*7*	0	0
5					*0*	0
>5						*1*
Σ	0	72	22	10	0	1
ΣΣ			105			
p		10^{-73}	10^{-13}	10^{-4}		

Column header: **#total statements**

too rare to derive statistically valid information, shown by gray areas of the last row in Tables 3.4 and 3.5.

We can summarize the results across micro-clone lengths into the two events "error not in last line or statement" and "error in last line or statement", shown in Table 3.3. Our absolute counts show that in micro-clones similar to Example 3.1, the last line is almost thrice as likely to contain a fault than all previous lines taken together. When looking at the individual line lengths in Table 3.4, the last line effect is even as high as a nine-fold increased error-proneness for the oft-appearing clone lengths 2, 4 and 5. The results for cloned statements in micro-clones within one line, like Example 3.2, are stronger still: We found the last statement to be 9.5 times more faulty than all other statements taken together. In fact, for the 72 micro-clones consisting of two statements, the last statement was the faulty one in all but one case.

> In a micro-clone, the last line is almost thrice as likely to be erroneous than all previous lines combined.

In total, our findings confirm the presence of a pronounced last line and last statement effect, accepting both RQ II.1 and RQ II.2.

3.3.4 Origin of Micro-Clones

Having found a large number of seemingly trivial micro-clone-related bugs in OSS projects, we were curious about the reasons for its presence. In RQ II.3, we therefore ask:

RQ II.3 What are the reasons for the existence of faulty micro-clones and the last line effect in particular?

In this section, we first analyze the origin of micro-clone instances, and examine which technical and psychological reasons might play a role for the existence of micro-clones.

Table 3.6 shows the results of the copy origin analysis broken down per clone length. For it, we disregarded micro-clones for which we could not agree on the order of their clones, leaving us with 245 out of 263 clone pairs.

Table 3.6: Clone Length (Horizontal) and Likely Clone Origin (Vertical).

origin \ clone length	1	2	3	4	5	6	7	8	9	>9
1	0	132	22	7	2	1	0	0	1	0
2		3	28	3	4	1	1	1	0	0
3			1	20	0	0	0	0	1	1
4				0	2	1	1	0	0	3
5					0	2	0	0	0	0
6						0	0	0	0	2
7							0	0	1	3
8								0	0	1
9									0	0
>9										0
Σ	0	135	51	30	8	5	2	1	3	10
ΣΣ	245									

In Figure 3.2, we plot the distribution of the copy origin. The figure shows that for 165 out of 245 micro-clones (67%), the first clone instance of a micro-clone is the influential one, with a large drop toward the second (18%) and subsequent gradual drops from the second to the third (9%) and fourth (3%). Only in the remaining 4% of cases does the influencing clone instance lie beyond the fourth line or statement in a micro-clone. This seems to indicate that the first line is most influential for the outcome of a clone. However, our distribution of micro-clones itself is highly skewed toward 2-liner micro-clones. It follows naturally that in most instances of a two-liner micro-clone, the origin lies in the first line.

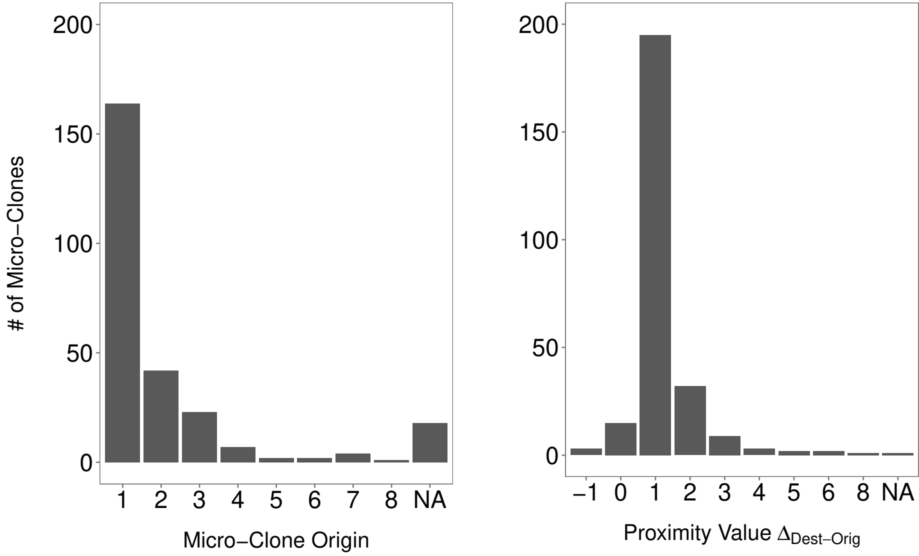


Figure 3.2: Copy Origin Distribution (left) and Proximity Distribution of Copy Origin and Destination within Micro-Clones (right).

When considering the 117 micro-clones which are longer than two clone instances in Table 3.6, we find that the copy origin is the first line only for 33 micro-clones (28%). As the average length for these 117 micro-clones is 4.9, we would expect 20% of copy origins to be in the first line, even for a uniform distribution. Our 28% indicate that the first line only exhibits a mild influence when correcting for the influence of 2-liner micro-clones. However, In 2-liner micro-clones, the first line is almost always the origin (rather than the second line influencing the first).

Figure 3.2 plots the distribution $\Delta_{Dest-Orig}$ (see Section 3.2.3). It shows that over 84% of clone instances appear in the immediate mutual neighborhood of each other (220 out of 245), i.e. $\Delta_{Dest-Orig} \leq 1$. In 89% of these cases (195 out of 220), $\Delta_{Dest-Orig} = 1$ or $\Delta_{Dest-Orig} = -1$, which means that the erroneous instance succeeds the correct instance in either the next line or statement. Preceding it, i.e. $\Delta_{Dest-Orig} = -1$, is much rarer (3 out of 220). When we disregard 2-liner or 2-statement clones, which naturally appear next to each other, we obtain that 81% of clone pairs appear in mutual neighborhood (66 out of 81). We can therefore summarize these findings with two general observations:

1. For 2-liner micro-clones, the first line is almost always the influencing origin. When correcting for the effects of short 2-liner code clones, in general, the first line of a micro-clone seems to only exhibit a mild additional influence as a source.
2. Instead, the influencing and the erroneous clone instance appear in direct textual and visual neighborhood in the source code in four out of five micro-clones. Moreover, in nine out of ten of these cases, the erroneous clone instance appears after its influencing origin.

> In most cases, copying errors seem to originate one line (or statement) above the erroneous instance.

3.3.5 Developer Interviews

In C_2, we approached ten authors of real-world committed micro-clones with excerpts of the micro-clone they authored and additional contextual information. We then asked them whether they remembered

1. how they mechanically created the micro-clone (e.g., by copy and pasting).
2. how the particular error referenced occurred or slipped-through.
3. which situation they were in when they created the commit, supplied with the local time and date of the commit.
4. in which stages of development and how often similar micro-clones are generally created in their experience.

Table 3.7 gives an overview of the ten micro-clones and associated seven interviews which we lead asynchronously via email and Skype. The table denotes the sampled projects and commits, the creation date of the commits, their median and individual sizes

Table 3.7: Descriptive Statistics of Developer Interviews and Commit Size Analysis of sampled repositories before 6.10.2016.

Project	Sampled Commit	Local Commit Date	Commit Churn	Median Churn	#Commits	Replies
Chromium	2db5310 6b7fcb4 (7b37fbb) 47fcb0e	2010-09-30 20:53 2011-02-23 05:57 (2011-03-07 16:16) 2012-10-24 3:52	123 1220 (1,635) 1,627	43	639,564	4/4
LibreOffice	b90bc10 44cfc7c (rebase)	2008-08-19 22:06 2012-10-09 12:22	103,083 470	18	438,994	0/2
Samba	781ed1f	2005-12-09 05:21	45	16	241,276	1/1
Mesa 3D	0ff3b2b 45124e0	2010-07-26 23:56 2010-12-07 21:37	108 251	21	99,115	1/2
libjingle	562554d	2010-09-30 20:53	110,184	212	341	1/1
$\sum$	10					7/10 (6 authors)

in terms of churn, and the total number of commits in the project. To protect the identity of interviewees *I1–I7*, we do not connect the IDs with commits in the table and also anonymize all subsequent code fragments. If we received no reply after one week, we sent a one-time reminder to participate to the interviewee. In the following, we summarize the insights we obtained from the interviews. We discuss interviews *I1*, *I2*, *I4*, *I6*, and *I7* at length. As we reached a preliminary saturation, our abbreviated findings here summarize the other remaining interviews.

One interviewee replied that he has "no interest." Another interview ended because the participant replied that the commit was too long ago and he does not remember it. In one instance, `7b37fbb`, the interviewee *I1* told us that he merely refactored and did not author this piece of code originally (hence we report six interviews with authors in Table 3.7). He forwarded us to the real author of the code, whom we also interviewed (`6b7fcb4`).

We asked *I2* about the micro-clone:

Example 3.14: Anonymized I1

```
1 if (!has_mic && !has_mic) {
```

He told us that the mistake was not a copy-and-paste mistake. Rather, he typed `!has_mic` when he should have typed `!has_audio` instead. In his experience, this happens a lot when working with code in which one types the same words repetitively. He observed that "I was not under any major stress at the time," but that "I will note that when working with very large changes it is much easier for something like this to be missed." He added that the real error was not having a unit test that covers this line and that the reviewer missed the absurdity of the pattern `!a && !a`, too.

I4 answered that, while he did not remember this case specifically, he reconstructed what likely happened for the micro-clone of the form:

Example 3.15: Anonymized I4

```
return
field.type == trans("text") ||
field.type == trans("twitter") ||
field.type == trans("mail") ||
field.type == trans("http") ||
field.type == trans("email") ||
field.type == trans("text");
```


When creating such micro-clones, he usually comes up with the first clone instance `field.type == trans("text") ||` and copy-and-pastes it several times, ending up in a sequence like:

Example 3.16: Anonymized I4

```
field.type == trans("text") ||
field.type == trans("text") ||
...
```

He reported that he does "not carefully count how many repetitions there are – I just guesstimate." As a last step, he would also remember to delete any extraneous lines, but he assumes that he did not remember in this case or got distracted. During the origin analysis (see Section 3.2.3), we also found that two refactorings on this micro-clone were performed, but the original error stayed. This happened because developers relied on a tool to do the transformation for them and did not read the code carefully. *I4* concluded that he often uses these mechanics for creating micro-clones, "but I usually remember to pare down any extraneous lines." Similar to *I1*, he also stated that it should be caught by either code review or testing.

I6 answered that "it has been a while, but [...] this seems like [a] copy/paste bug to me. Not uncommon." He also said "I see (and do) this kind of thing all the time." To move fast and save typing, *I6* created the micro-clone by copy-and-pasting, then modifying each line by varying it. "The last line got missed." His explanation is that he forgot to modify the last micro-clone instance, since "usually, my mind has moved on to less mechanical thought. But then the mechanical action gets botched." While *I6* did not recall the day particularly, they are "always trying to move fast to get improvements out." He also said that he sees micro-clones "all the time," at least 10 times per day. "Of those 10, perhaps 9 get caught in self review or by the compiler. The last one gets caught by other reviewers or unit tests mostly. But on occasion, say once a month [...], this kind of [bug] makes it into shipping code that affects end users."

I7 authored a micro-clone of the format

Example 3.17: Anonymized I7

```
else if(depth > 0 && width > 0 && width > 0)
```

He remembered that he "just typed it out, no copy/paste" and missed it because "I was probably in a hurry and was not focused." While he could not remember the specific date, he noted that he is "always pretty busy in general."

From the interviews, it seemed that one factor that might affect the likelihood of faulty micro-clones to pass through the various measures of defense the interviewees mentioned, might be the size of the commit. If this is the case, then micro-clone inducing commits should be abnormally large. The term "abnormally large" only makes sense in the context

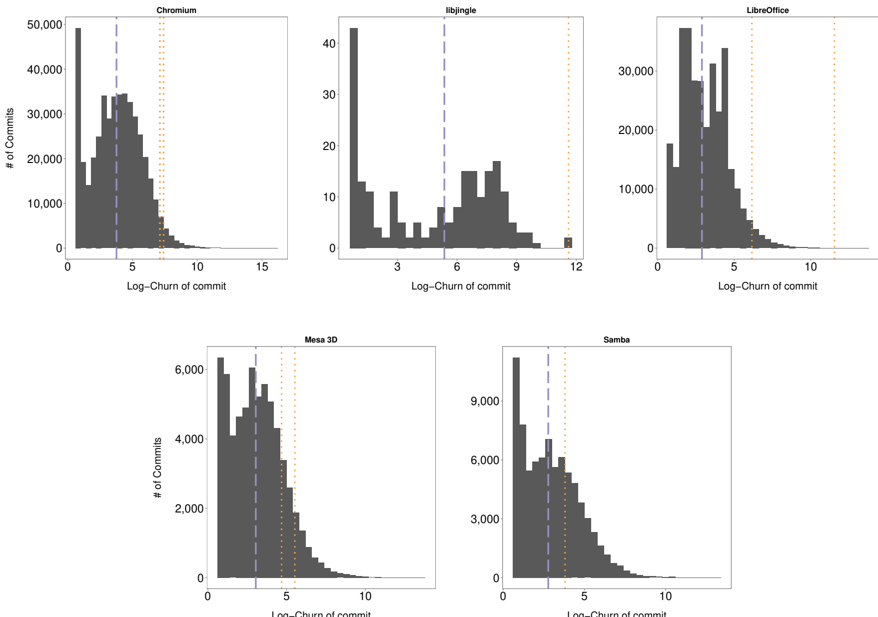


Figure 3.3: Median commit size over whole repository history (dashed blue) and commit size (as logarithmic churn) of individual micro-clone introducing commits (dotted orange).

of each project's relative commit sizes. In Figure 3.3, we therefore compare the size of the sampled micro-clone inducing commits to the median commit size in each project. The figures show that all micro-clone inducing commits were orders of magnitude larger than the median commit sizes in each project.

3.3.6 Usefulness of Results

Having unveiled a large number of potential bugs in OSS, we wanted to help the OSS community and see if our findings represented bugs that would be worth fixing in reality. We approached the OSS projects by creating issues with our findings in their bug trackers. Many of our bug reports lead to quality improvements in the projects, like fixing the validation bug from Example 3.2 in Chromium.[9] The search query `pvs-studio bug | issue`[10] shows numerous bug fixes in Firefox, libxml, MySQL, Clang, samba and many other projects based on our findings. As one such example, on October 11th 2016 in commit `caff670`, we fixed a micro-clone-related issue that had existed in samba since 2005.[11]

3.4 Discussion

In this section, we discuss our results by merging the observed bug patterns with our psycho-cognitive analysis. We end with an explanation of possible threats to the validity

[9] https://codereview.chromium.org/7031055
[10] www.google.com/search?q=pvs-studio+bug+\TU\textbar+issue
[11] https://bugzilla.samba.org/show_bug.cgi?id=12373

of our conclusions.

3.4.1 Technical Complexity & Reasons

Technical reasons that could play a role for the existence of the last line effect would assume that the last line in a micro-clone is technically more complex in comparison to the other lines, and thus more likely to contain an error. This would include that the last line is for example not checked by the compiler, or that, when an Integrated Development Environment (IDE) is used and the last line indeed written as the last action in this editor window, perhaps the compiler would not react fast enough to check it. This is not true for two reasons:

1. Modern IDEs typically do not lag behind in syntax checking.
2. The last line or statement micro-clone instances are grammatical, i.e. a compiler error that would draw attention to them does not occur.

On the other hand, if IDEs and compilers did include checks for micro-clones, they could help developers catch them before committing.[12]

Another technical reason might be that coming up with the last statement in a series of statements might be harder than the ones before. However, when observing any of the Examples 3.1, 3.2, 3.5, 3.7 and 3.11, it becomes clear that the opposite is the case: Because all clone instances in a micro-clone follow the same pattern, the hardest to come up with, if any, is the first instance. The succeeding instances simply replicate its pattern.

3.4.2 Psychological Mechanisms & Reasons

As technical reasons are not a likely cause for the last line effect, we consider here psychological mechanisms that might underlie this effect. We turned to a professor in cognitive psychology (the fourth author of this chapter) and presented our findings to him. In the following, we give an initial overview of possible psychological effects. These psychological reasons are preliminary at this point, because a more detailed analysis would require psychological experimentation in which the actual process of producing these errors is observed, rather than reconstructed by an origin analysis (see Section 3.2.3) and remembered in interviews (see Section 3.3.5).

In cognitive psychology, action slips are errors that occur during routine tasks and have been widely studied [150]. A typical example would be to put milk in a coffee twice instead of milk and sugar. Our analysis on the origin of micro-clones concluded that developers follow a wide variety of different mechanical patterns to create micro-clones. One of these patterns is "[write first clone instance], [copy], [copy], ..., [modify], [modify], ...", see *I4, I6.* Our interviews and origin-analysis also show that developers equally follow the pattern "[write first clone instance], [copy, modify], [copy, modify], ..." In some extreme cases micro-cloning in our data set, this action sequence must have been repeated 34 times. Even though they use different mechanical methods, the task software developers are performing in producing micro-clones can always be seen as a sequential action task with different levels of automation and manual effort. From a psycho-cognitive viewpoint, er-

[12] Clang started to implement our analyses, see `https://llvm.org/bugs/show_bug.cgi?id=9952`.

rors developers introduce while producing micro-clones are thus characterized as typical action slips.

While differing on the details, models for sequential action control assume that noise is the main explanation for action slips [151–153]. By *noise*, we refer to any task-irrelevant mental representation, which includes external stress such as deadlines and internal factors such as large commits, that might draw the developer's attention. Sequential action control models provide a useful theoretical framework for speculating about the psychological mechanisms behind the last line effect. At this point, we only know the faulty micro-clones instances, and their location, but we have no detailed process information on how they came to be. However, as Section 3.3.4 showed, the anecdotal evidence from interviews as well as our technical origin analysis does allow us to make informed inferences about the creating of an erroneous micro-clone instance. The basic operations that the programmer performs are: copying and editing. Consider Example 3.1 again. The editing step here involves two sub-steps, updating the variable and updating the value.

Example 3.1: TrinityCore

```
1 x += other.x;
2 y += other.y;
3 z += other.y;
```

Here, line 3 contains an error. It appears that line 2 was copied to produce line 3. The first update was performed correctly (change y into z) but the second editing sub-step was not performed, thus producing the error. In principle, line 1 could have been copied twice with the editing steps having been performed on the two lines. However, the presence of a y rather than x in line 3 suggests that line 2 was copied. Section 3.3.4 shows that in most cases of micro-clones with more than two lines, the previous line was copied. This suggests that in such micro-clones, the sequence of actions was as follows: "[copy, modify, modify], [copy, modify, modify], ..."

Models of action control assume that action slips occur because of noise. Such noise is more likely to occur near the end of a sequence because the programmer's focus might prematurely shift to the next task, for example subsequent lines of code that need to be produced (see evidence from *I6*). As noted earlier, there are subtly different psychological explanations for why such noise might occur. To take just one account [152], the last line effect might occur because the wrong action schema is selected (e.g., the engineer is already mentally working on the next lines rather than completing the current one).

Although none of the engineers noted to have experienced extraordinary stress levels at the time of the creation of the clone, the statements from *I6* and *I7* stand out, who indicated a general sense of business and desire to move fast. When considering the local commit dates of when erroneous micro-clones in Table 3.7, it stands out that only two were created during core office hours, even though many interviewees did this as part of their job. Tiredness is known to reduce brain efficiency and affect the working memory [154]. This could indicate that tiredness and a general time pressure might play a critical role in the creation of erroneous micro-clones.

More than time pressure, however, we found that all micro-clone inducing commits (and even refactorings) were exceptionally large – orders of magnitude larger than a normal commit in the repositories. We therefore purport that commit size is an important, perhaps the dominant noise factor, that makes these errors go unnoticed. This finding cor-

roborates well with the explanations of a working memory overload and *I1's* observation that the resulting amount of code is very hard to oversee.

Our interviews with developers indicated that creating short-lived micro-clones might be common in software development, but that the developers usually catch them early, or at least during their own or someone else's review of the code [27]. The cognitive error in the remaining micro-clones we observed in this study is thus not only a production error, but also a proofreading error [155]: During revision of the code, the engineer fails to notice the error in the last and other lines. In fact, our interviews suggest that this seemed to happen twice for the micro-clones that made to production: once, during self-review and then at least one second time during code review by a peer. One plausible reason why this proofreading error is more likely to occur in the last line than in earlier ones could be because it is an action slip. The mind is already focused on the next task (e.g., implementing a new feature) before the current one (proofreading), has been completed. Yet another account could be that the error is less detectable because several very similar statements in a row have to be proofread. This could cause the reading of the final statement to be faster and therefore more superficial. Moreover, the visual closeness of origin and target in micro-clones might simply make it more difficult to differentiate between the individual lines. Research on proofreading suggests that familiarity (operationalized as word frequency) leads to shorter processing times and has a negative impact on the ability to detect spelling errors in text [156].

All potential causes suggest that developers are more likely to conduct last-line-type errors in situations when their attention span is reduced through noise. Possible causes for noise with a negative impact on micro-cloning in particular seem to be large commit sizes, and possibly high workload, stress, being distracted, and tiredness [157]. Conversely, our results also suggest that developers' ability to control irrelevant noise from the environment [158], i.e. their ability to focus attention, plays an important role in how likely a micro-clone is going to be created with an action slip related error.

3.4.3 Threats to Validity

In this section, we show internal and external threats to the validity of our results, and how we mitigated them.

Internal Threats

An important internal threat to this study concerns how to determine in which line the error lies. Given Example 3.2, any of the two statements could be counted as the one containing the duplication. However, reading and writing source code typically happens from top to bottom and from left to right [7]. Therefore, the only natural assessment is to flag lines and statements as problematic according to this strict left-right and top-down visual reading order: In Example 3.2, only when we have read the second statement do we know it is a duplicate of the first. We hence flag the second statement as the one containing the error. Moreover, in many cases, as in Example 3.2, the program context around the micro-clone (here the definition order of the variables `host` first and then `port_str`) imposes a natural logical order for the remainder of the program (first check `host`, then `port_str` in line 3). In order to reduce personal bias, we also separated the list of findings to triage across the first two authors, and then discussed unclear cases. If we could

not reach agreement, we discarded said finding. In this process, we also re-classified all original previous 202 findings [63] and found almost total agreement with our previous assessment. Since flagging erroneous lines is a well-defined task under these circumstances, we are sure there is a high inter-rater reliability, ensuring the repeatability of our study.

It is likely that our checkers are not exhaustive in detecting all faulty micro-clones. This poses only a small threat to the validity of our results, as we do not claim to cover all micro-clones. We believe to have captured a major part of the micro-clone population through extending our checkers to 12 (see Table 3.1). Evidence that our analyses capture a major source of bugs comes from the fact that only our core checkers V501, V517, V519 and V537 add a substantial share of the results and that van Tonder and Le Goues found more than 24,000 faulty micro-clones using a subset of our checkers [159].

While we are confident about the results of our origin-destination analysis, we do not know how the clones were created and modified by the software developer. Our a-posteriori repository mining approach assumes a top-to-bottom reading order of blocks and a left-to-right reading order for individual lines. We know that developers “jump” in the code when reading a file, only focusing on what seems important to solve the task at hand [7, 160]. However, in order to understand small coherent logical units, such as micro-clones, developers must necessarily read in the control-flow-direction of the software – which is top-to-bottom, left-to-right. In particular, it would be interesting to see 1) how many times the copy-paste-pattern “`ctrl+c`, `ctrl+v`” was used, 2) in which order micro-clones are created, and 3) in which order micro-clones are read and changed, if developers need to modify them during maintenance. In order to get to know such information, we would require to study how developers work in-vivo, similar to the WatchDog plugin [65–67]. To that end, we could reuse parts of CloneBoard, which captures all cut, copy and paste actions in Eclipse [161].

Given these limitations, our psychological analysis is partly speculative at this point. A more detailed analysis requires psychological experimentation in which the process of producing these errors is examined in-vivo. With our choice of research methods, we might potentially miss subtle steps in the creation of micro-clones. However, we believe that it is very difficult to expose faulty micro-cloning in a laboratory setting, as our interviews indicate that it requires a long time to expose a relatively small number of micro-clones. Moreover, due to the artificial nature of the experiment, a possible time limit and the fact that participants typically over-perform in experiments [51], they might not create faulty micro-clones at all. Since the results of our mixed-methods case studies corroborate each other, we believe to have acquired an accurate set of initial reasons for the existence of faulty micro-clones and the last line effect in particular.

External Threats

An external factor that threatens the generalizability is that PVS-Studio is specific to C and C++. C is one of the most commonly used languages [162]. Therefore, even if our results were not generalizable, they would at least be valuable to the large C and C++ communities. However, our findings typically contain language features common to most programming languages, such as the variable assignments, if clauses, Boolean expressions and array uses in Examples 3.1, 3.2, 3.5, 3.7 and 3.11. Almost all programming languages have these constructs. Thus, we expect to see analogous results in at least C-inspired

languages such as Java, JavaScript, C#, PHP, Ruby, or Python. While our overall corpus of findings is large, the average number of ~1.2 micro-clones per project is rather small (see Table 3.2). This could be because PVS-Studio's analyses for defective micro-clones are not exhaustive, and that the projects we studied are stable, production systems with a mature code base containing relatively little trivial errors. Our interviews gave another explanation: extensive testing and code-reviewing significantly decreases the number of faulty micro-clones that make it into production.

3.5 Related Work

Duplicated or similar code fragments are famously known as "code clones," yet their definition has remained somewhat vague over the last decade [163]. This vagueness is reflected in the definitions "[c]lones are segments of code that are similar according to some definition of similarity" by Baxter et al. [164] and "code clones [...] are code fragments of considerable length and significant similarity" by Basit and Jarzabek [165]. The latter definition identifies clones as long enough pieces of code that share sufficiently many traits, while the first has no such requirements. A widely-used definition categorizes clones into three classes [148]: Type 1 clones are textually and type 2 clones syntactically (modulo identifier renamings) identical. Type 3 clones have further-reaching syntactic modifications and type 4 clones are only functionally identical [163]. However, this general classification is agnostic about, for example, code clone length. Subsequently, researchers developed a plethora of more specific clone definitions [148, 166, 167]. In this study, we add to these taxonomies the concept of very short, but closely related code clones that are located below the lower limit of "considerable length," with often no more than two duplicated statements within one clone instance. We call such extremely short duplicated pieces of code, *micro-clones*.

In the following, we compare traditional code clone detection mechanisms to how we detect micro-clones. In a 2007 comparison and evaluation of clone detection tools, Bellon et al. evaluated six clone detectors for C and Java [145]. Depending on the clone detector, clones had to be at least six lines or 25 tokens long in their experiment. In 2014, Svajlenko and Roy performed a similar study and compared the recall performance of eleven modern clone detection tools [168]. In their configuration of the clone detectors, they used minimal clone lengths of 50 tokens, 15 statements, or 15 lines [168]. These thresholds are too large to be able to detect micro-clones. However, traditional clone detectors need them to avoid a large number of false positives. Our approach circumvents this problem by only detecting faulty micro-clones.

In direct follow-up research on our initial investigation [63], van Tonder and Le Goues performed a large-scale search for micro-clones in 380,125 Java repositories [159]. They found 24,304 faulty micro-clones, demonstrating and solidifying our assumption that micro-clones are a wide-spread phenomenon across software projects. By providing 43 patches to fix faulty micro-clones of which 42 were promptly integrated, they show that developers value the removal of micro-clones and that it can be automated at scale.

Empirical investigations with traditional clone detectors suggest that ~9% to 17% is a typical portion of clones in the code base of software systems [169], considering all type 1, 2, and 3 clones [148]. Outliers in the so-called "clone coverage" may be lower than 5% [170] and higher than 50% [163, 171]. These ratios do not include micro-clones, which

we have shown to be a frequent source of bugs in numerous OSS in this study. In the larger perspective of how prevalent code clones are in systems, micro-clones might lead to an increased perception of clones in the code, and to a much higher clone coverage, at least when considering a "micro-clone coverage" measure. A high clone coverage is generally thought to be problematic, since numerous studies have shown that it is positively correlated with bugs and inconsistencies in the system [172–175].

3.6 Future Work & Conclusion

In this section, we describe possible extensions of our study and draw conclusion.

Because our study focuses on faulty micro-clones, we cannot make predictions about how many of all micro-clones are erroneous. A promising future research direction is to develop a clone detector that can reliably detect all micro-clones, and then to see how many are actually defective. This gives an indication of the scale of the problem at hand. Anecdotal evidence from interviews suggests that micro-cloning seems to happen quite often and catching it consumes precious code review and testing iterations. To catch faulty micro-clones early, including our checkers for micro-clones into the IDEs of developers seems to be fruitful direction for future work.

Our initial psychological examination of the effect warrants a larger psychological controlled experiment, that, we believe, might be associated with a high risk of not exposing enough faulty micro-clone creations. Already existing tooling could help enable this study on a technical level.

In 219 open source projects, we found 263 faulty micro-clones. Our analysis shows that there is a strong tendency for the last line, and an even stronger tendency for the last statement to be faulty, called the last line effect. In fact, the last line in a micro-clone is three times as likely to contain a fault than any of the previous lines combined, and the last statement almost ten times as likely as any of the previous statements combined.

Psychological reasons for the existence of the last line effect seem to be largely the result of *action slips*, where developers fail to carry out a repetitive and easy process correctly, caused by working memory overload by noise. We have evidence suggesting that the effect is largely caused by the way developers copy-and-paste code. Developers seem to have a psychological tendency to think changes of similar code blocks are finished earlier than they really are. This way, they miss one critical last modification. Important reasons for noise seem to be abnormally large commit sizes, and possibly tiredness and stress.

Because of this observation, we advise programmers to be extra-careful when reading, modifying, creating, or code-reviewing the last line and statement of a micro-clone, especially when they copy-and-paste it. Moreover, our finding that faulty micro-clones were only present in abnormally large commits emphasizes the importance of small, manageable commits. This knowledge can help developers alleviate bugs due to faulty micro-clones, while writing and reviewing code. Developers can spot mental situations in which they are likely to commit errors due to working term memory overload and pay attention to avoid them. With PVS-Studio, we have described an ASAT that supports developers to spot when such errors have "slipped through" pre-release, for example during code review.

4

Developer Testing in the IDE: Patterns, Beliefs, and Behavior

Software testing is one of the key activities to achieve software quality in practice. Despite its importance, however, we have a remarkable lack of knowledge on how developers test in real-world projects. In this chapter, we report on a large-scale field study with 2,443 software engineers whose development activities we closely monitored over 2.5 years in four integrated development environments (IDEs). Our findings, which largely generalized across the studied IDEs and programming languages Java and C#, question several commonly shared assumptions and beliefs about developer testing: half of the developers in our study do not test; developers rarely run their tests in the IDE; most programming sessions end without any test execution; only once they start testing, do developers do it extensively; a quarter of test cases are responsible for three quarters of all test failures; 12% of tests show flaky behavior; Test-Driven Development (TDD) is not widely practiced; and software developers only spend a quarter of their time engineering tests, whereas they think they test half of their time. We compile these practices of loosely guiding one's development efforts with the help of testing in an initial summary on Test-Guided Development (TGD), a behavior we argue to be closer to the development reality of most developers than TDD.

This chapter is to appear as *M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman: Developer Testing in The IDE: Patterns, Beliefs, And Behavior, TSE* [64], an extension of

- *M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, How, and Why Developers (Do Not) Test in Their IDEs, ESEC/FSE'15* [66],
- *M. Beller, G. Gousios, and A. Zaidman. How (Much) Do Developers Test? ICSE'15 (NIER)* [65], and
- *M. Beller, I. Levaja, A. Panichella, G. Gousios, and A. Zaidman. How to Catch 'em All: WatchDog, a Family of IDE Plug-Ins to Assess Testing, SER&IP'16* [67].

How much should we test? And when should we stop testing? Since the beginning of software testing, these questions have tormented developers and their managers alike. In 2006, twelve software companies declared them pressing issues during a survey on unit testing by Runeson [176]. Fast-forward to eleven years later, and the questions are still open, appearing as one of the grand research challenges in empirical software engineering [177]. But before we are able to answer how we *should* test, we must first know how we *are* testing.

Post mortem analyses of software repositories by Pinto et al. [178] and Zaidman et al. [179] have provided us with insights into how developers create and evolve tests at the commit level. However, there is a surprising lack of knowledge of how developers *actually* test, as evidenced by Bertolino's and Mäntylä's calls to better understand testing practices [47, 180]. This lack of empirical knowledge of when, how, and why developers test in their Integrated Development Environments (IDEs) stands in contrast to a large body of folklore in software engineering [177], including Brooks' statement from "The Mythical Man Month" [181] that "testing consumes half of the development time."

To replace folklore by real-world observations, we studied the testing practices of 416 software developers [66] and 40 computer science students [65] with our purpose-built IDE plugin WatchDog. While these studies started to shed light on how developers test, they had a number of limitations toward their generalizability: First, they were based on data from only one IDE, Eclipse, and one programming language, Java. It was unclear how the findings would generalize to other programming environments and languages. Second, the data collection period of these studies stretched only a period of five months. This might not capture a complete real-world "development cycle," in which long phases of implementation-heavy work follow phases of test-heavy development [179, 182]. Third, we did not know how strongly the incentives we gave developers to install WatchDog influenced their behavior. Fourth, we had no externally collected data set to validate our observations against.

In this extension of our original WatchDog paper [66], built on top of our initial draft of the WatchDog idea [65] and its technical tool description [67], we address these limitations by analyzing data from four IDEs, namely Eclipse (EC), IntelliJ (IJ), Android Studio (AS), and Visual Studio (VS), and two programming languages, Java and C#. We extended our study from 416 developers to an open-ended field study [183] with 2,433 developers that stretches over a data collection period of 2.5 years. By measuring how developers use the behavior reports WatchDog provides as an incentive, we can now estimate their impact on developers' behavior. Thanks to Visual Studio data from the FeedBaG++ plugin, developed independently in the KaVE project [184], we can compare our findings against an externally collected data set.

In our investigation, we focus on developer tests [185], i.e., codified unit, integration, or system tests that are engineered inside the IDE by the developer. Developer testing in the IDE is often complemented by work outside the IDE, such as testing on the CI server [68], executing tests on the command line, manual testing, automated test generation, and dedicated testers, which we explicitly leave out of our investigation. By comparing the *state of the practice* to the *state of the art* of testing in the IDE [46, 186, 187], we aim to understand the testing patterns and needs of software engineers, expressed in our five research questions:

RQ III.1 Which Testing Patterns Are Common In The IDE?

RQ III.2 What Characterizes The Tests Developers Run In The IDE?

RQ III.3 How Do Developers Manage Failing Tests In The IDE?

RQ III.4 Do Developers Follow Test-Driven Development (TDD) In The IDE?

RQ III.5 How Much Do Developers Test In The IDE?

If we study these research questions in a large and varied population of software engineers, the answers to them can provide important implications for practitioners, designers of next-generation IDEs, and researchers. To this end, we have set up an open-ended field study [183] that has run for 2.5 years and involved 2,443 programmers from industry and open-source projects around the world. The field study is enabled by the Eclipse and IntelliJ plugin WatchDog and the Visual Studio plugin FeedBaG++, which instrument the IDE and objectively observe how developers work on and with tests.

Our results indicate that over half of the studied users do not practice testing; even if the projects contain tests, developers rarely execute them in the IDE; only a quarter of test cases is responsible for three quarters of all test failures; 12% of test cases show flaky behavior; Test-Driven Development is not a widely followed practice; and, completing the overall low results on testing, developers overestimate the time they devote to testing almost twofold. These results counter common beliefs about developer testing and could help explain the observed bug-proneness of real-world software systems.

4.1 Study Infrastructure Design

In this section, we give a high level overview of our field study infrastructure design, explore how a practitioner uses WatchDog to convey an intuitive understanding of the plugin, and describe how our plugins instrument the IDE.

4.1.1 Field Study Infrastructure

Starting with an initial prototype in 2012, we evolved our IDE instrumentation infrastructure around WatchDog into an open-source, multi-IDE, and production-ready software solution [188]. As of version 1.5 released in June 2016, it features the three-layer architecture depicted in Figure 4.1 with a client, server, and data analysis layer, designed to scale up to thousands of simultaneous users. In the remainder of this section, we first describe the client layer containing the four different IDE plugins for Visual Studio, IntelliJ, Android Studio and Eclipse (from left to right). We then describe WatchDog's server and central database and how we converted the KaVE project's FeedBaG++ data to WatchDog's native interval format. We conclude this high-level overview of our technical study design with a short description of our analysis pipeline. In earlier work, we have already given a more technical description of WatchDog's architecture and the lessons we learned while implementing it [67].

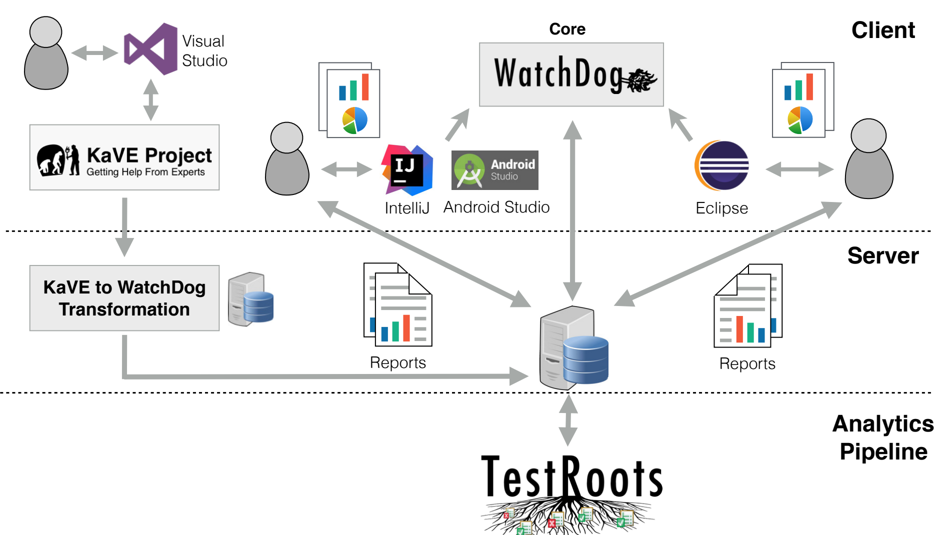


Figure 4.1: WatchDog's Three-Layer Architecture.

IDE Clients

We used two distinct clients to collect data from four IDEs: the WatchDog plugin gathers Java testing data from Eclipse and IntelliJ-based IDEs and the general-purpose interaction tracker FeedBaG++ gathers C# testing data from Visual Studio.

WatchDog clients for Eclipse and IntelliJ. We originally implemented WatchDog as an Eclipse plugin, because the Eclipse Java Development Tools edition (JDT) is one of the most widely used IDEs for Java programming [189]. With WatchDog 1.5, we extended it to support IntelliJ and IntelliJ-based development platforms, such as Android Studio, "the official IDE for Android" [190]. Thanks to their integrated JUnit support, these IDEs facilitate developer testing.

WatchDog instruments the Eclipse JDT and IntelliJ environments and registers listeners for user interface (UI) events related to programming behavior and test executions. Already on the client side, we group coherent events as *intervals*, which comprise a specific type, a start and an end time. This abstraction allows us to closely follow the workflow of a developer without being overwhelmed by hundreds of fine-grained UI events per minute. Every time a developer reads, modifies, or executes a JUnit test or production code class, WatchDog creates a new interval and enriches it with type-specific data.

FeedBaG++ for Visual Studio. FeedBaG++ is a general-purpose interaction tracker developed at TU Darmstadt. It is available for Visual Studio, as an extension to the widely used ReSharper plugin [191], which provides static analyses and refactoring tools to C# developers.

FeedBaG++ registers listeners for various IDE events from Visual Studio and the ReSharper extension, effectively capturing a superset of the WatchDog listeners. The captured information relevant for this chapter includes how developers navigate and edit source files and how they use the test runner provided by ReSharper. The test recogni-

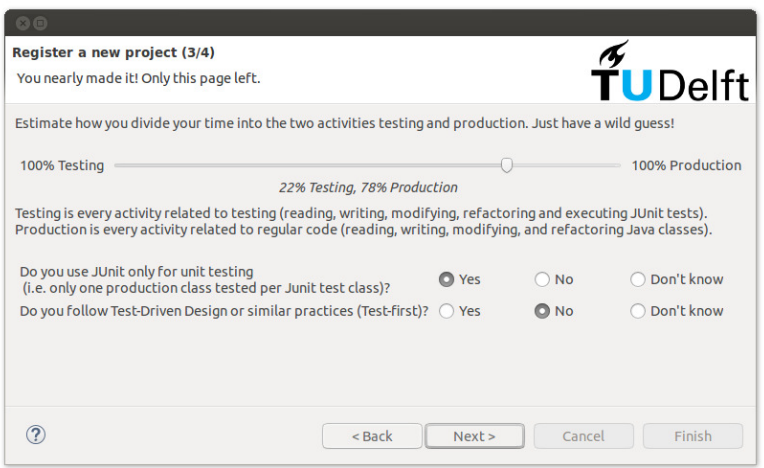


Figure 4.2: Exemplary wizard page of WatchDog's project survey.

tion covers common .NET testing frameworks, such as NUnit or MSUnit. In contrast to WatchDog, which already groups events into intervals on the client side, FeedBaG++ uploads the raw event stream.

WatchDog Server

The WatchDog IDE plugins cache intervals locally, to allow offline work, and automatically send them to our server as a JSON stream. The WatchDog server accepts this JSON data via its REST API. After sanity checking, the intervals are stored in a NoSQL database. This infrastructure scales up to thousands of clients and makes changes in the clients' data format easy to maintain. Moreover, we can remotely trigger an update of all WatchDog clients, which allows us to fix bugs and extend its functionality after deployment. Automated ping-services monitor the health of our web API, so we can immediately react if an outage occurs. Thereby, our WatchDog server achieved an average uptime of 98% during the 2.5 years of field study.

WatchDog Analysis Pipeline

The WatchDog pipeline is a software analytics engine written in R comprising over 3,000 source lines of code without whitespaces (SLOC). We use it to answer our research questions and to generate daily reports for the WatchDog users. The pipeline reads in WatchDog's users, projects, and intervals from the NoSQL database and converts them into intermediate formats fit for answering our research questions.

4.1.2 WatchDog Developer Survey & Testing Analytics

To give an understanding of the study context and incentives that WatchDog offers, we explore it from a practitioner's perspective in this section. Wendy is an open-source developer who wants to monitor how much she is testing during her daily development activities inside her IDE. Since Wendy uses IntelliJ, she installs the WatchDog plug-in from the IntelliJ plug-in repository.

Registration. Once installed, a wizard guides Wendy through the WatchDog registration process: First, she registers herself as a user, then the project for which WatchDog should collect development and testing statistics, and finally, she fills in an interactive voluntary in-IDE survey about testing. Figure 4.2 shows one of the up to five pages of the survey. Key questions regard developers' programming expertise, whether and how they test their software, which testing frameworks they employ and how much time they think they spend on testing. Since FeedBaG++ does not collect comparable survey data, we exclude it from research questions relying on it. Wendy, however, continues to work on her project using IntelliJ, as usual, while WatchDog silently records her testing behavior in the background.

Developer Statistics. After a short development task, Wendy wants to know how much of her effort she devoted to testing and whether she followed TDD. She can retrieve two types of analytics: the *immediate statistics* inside the IDE shown in Figure 4.3 and her personal *project report* on our website shown in Figure 4.4. Wendy opens the *immediate statistics* view. WatchDog automatically analyzes the recorded data and generates the view in Figure 4.3, which provides information about production and test code activities within a selected time window. Sub-graph ① in Figure 4.3 shows Wendy that she spent more time (over one minute) reading than writing (only a few seconds). Moreover, of the two tests she executed ②, one was successful and one failed. Their average execution runtime was 1.5 seconds. Finally, Wendy observes that the majority (55%) of her development time has been devoted to engineering tests ③, not unusual for TDD [66].

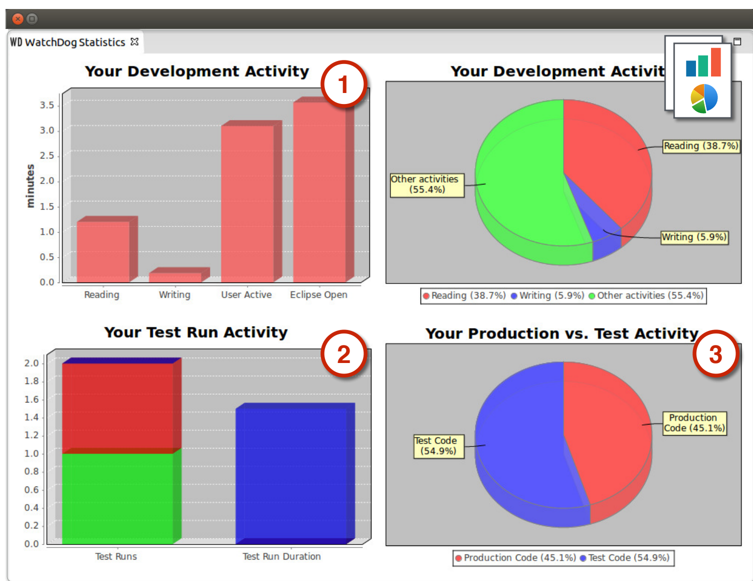


Figure 4.3: WatchDog's Immediate Statistics View in the IDE (Source: [67]).

While the *immediate statistics* view provides Wendy with an overview of recent activities inside the IDE, the *project report* gives her a more holistic view of her development behavior, including more computationally expensive analyses over the whole project his-

Detailed Statistics

In the following table, you can find more detailed statistics on your project.

Description	Your value	Mean
Total time in which WatchDog was active	195.8h	79h
Time averaged per day	0.6h / day	4.9h / day
General Development Behavior	**Your value**	**Mean**
Active Eclipse Usage (of the time Eclipse was open)	58%	40%
Time spent Writing	13%	30%
Time spent Reading	11%	32%
Java Development Behaviour	**Your value**	**Mean**
Time spent writing Java code	55%	49%
Time spent reading Java code	45%	49%
Time spent in debug mode	0% (0h)	2h
Testing Behaviour	**Your value**	**Mean**
Estimated Time Working on Tests	50%	67%
Actual time working on testing	44%	10%
Estimated Time Working on Production	50%	32%
Actual time spent on production code	56%	88%
Test Execution Behaviour	**Your value**	**Mean**
Number of test executions	900	25
Number of test executions per day	3/day	1.58/day
Number of failing tests	370 (41%)	14.29 (57%)
Average test run duration	0.09 sec	3.12 sec

2

Summary of your Test-Driven Development Practices
You followed Test-Driven Development (TDD) 38.55% of your development changes (so, in words, quite often). With this TDD followship, your project is in the top 2 (0.1%) of all WatchDog projects. Your TDD cycle is made up of 64.34% refactoring and 35.66% testing phase.

Figure 4.4: WatchDog's Project Report (Source: [67]).

tory. She accesses her report through a link from the IDE or directly via the TestRoots website,[1] providing the project's ID. Wendy's online project report summarizes her development behavior in the IDE over the whole recorded project lifetime. Reading the report in Figure 4.4, Wendy observes that she spent over 195 hours in total on the project under analysis, an average of 36 minutes per day ①. She worked actively with IntelliJ in 58% of the time that the IDE was actually open. The time spent on writing Java code corresponds to 55% of the total time, while she spent the remaining 45% reading Java code. When registering the project, Wendy estimated the working time she would spend on testing to equal 50%. With the help of report, she finds out that her initial estimation was relatively precise, since she actually spent 44% of her time working on test code.

The *project report* also provides Wendy with *TDD statistics* for the project under analysis, ② in Figure 4.4. Moreover, anonymized and averaged statistics from the large WatchDog user base allow Wendy to put her own development practices into perspective. This way, project reports foster comparison and learning among developers. Wendy finds that, for her small change, she was well above average regarding TDD use: She learned how to develop TDD-style from the "Let's Developer" YouTube channel.[2] The WatchDog project for "Let's Developer" is the second highest TDD follower of all WatchDog users on 5th June, 2017 (following TDD for 37% of all modifications).[3]

4.1.3 IDE Instrumentation

Here, we explain how WatchDog clients instrument the IDE. We then continue with a description of how we transform FeedBaG++ events into WatchDog intervals.

WatchDog Clients

WatchDog focuses around the concept of intervals. Table 4.1 gives a technical description of the different interval types. They appear in the same order as rows in Figure 4.5, which exemplifies a typical development workflow to demonstrate how WatchDog monitors IDE activity with intervals.

Exemplary Development Workflow. Our developer Wendy starts her IDE. The integrated WatchDog plugin creates three intervals: `EclipseOpen`, `Perspective`, and `UserActive` ①. Thereafter, Wendy executes the unit tests of the production class she needs to change, triggering the creation of a `JUnitExecution` interval, enriched with the test result "Passed" ②. Having browsed the source code of the file ③ to understand which parts need to change (a `Reading` interval is triggered), Wendy performs the necessary changes. A re-execution of the unit test shows Wendy that there is a failing test after her edit ④. Wendy steps through the test with the debugger ⑤ and fixes the error. The final re-execution of the test ⑥ succeeds.

Interval Concept. WatchDog starts or prolongs intervals concerning the user's activity (`Reading`, `Typing`, and other general activity) once it detects an interval-type preserving action. For example, if there is a `Reading` interval on class X started for 5 seconds and the plugin receives a scroll event, the interval is prolonged. However, if we detect that the IDE lost focus (end of `EclipseActive` interval), or the user switched from reading

[1] `http://testroots.org/report.html`
[2] `http://www.letsdeveloper.com`
[3] Project report: `http://goo.gl/k9KzYj`

Table 4.1: Overview of WatchDog intervals and how we transformed FeedBaG++ events to them. Related intervals appear without horizontal separation.

Interval Type	WatchDog Description	FeedBaG++ Transformation
`JUnitExecution` †	Interval creation invoked through the Eclipse JDT-integrated JUnit runner, which also works for Maven projects (example in Figure 4.6). Each test execution is enriched with the SHA-1 hash of its test name (making a link to a `Reading` or `Typing` interval possible), test result, test duration, and child tests executed.	FeedBaG++ tracks the ReSharper runner for the execution of NUnit tests. The results of running tests are easy to match to JUnit's result states. However, NUnit does not differentiate between *errored* and *failed* tests, so we map all failing runs to the latter and only report errors for *inconclusive* test runs.
`Reading`	Interval in which the user was reading in the IDE-integrated file editor. Enriched with an abstract representation of the read file, containing the SHA-1 hash of its filename, its SLOC, and whether it is production or test code. A test can further be categorized into a test (1) which uses JUnit and is, therefore, executable in the IDE, (2) which employs a testing framework, (3) which contains "Test" in its filename, or (4) which contains "test" in the project file path (case-insensitive). Backed by inactivity timeout.	FeedBaG++ tracks document and window events, allowing us to identify when a developer opens a specific file or brings it back to focus. If no other activity interrupts this, we count it as reading, until the inactivity threshold is reached.
`Typing`	Interval in which the user was typing in the IDE. Enriched with the Levenshtein edit distance, backed by inactivity timeout.	We use FeedBaG++'s edit events to distinguish `Reading` from `Typing` intervals and approximate the Levenshtein distance via the number of `Typing` intervals.
`UserActive`	Interval in which the user was actively working in the IDE (evidenced for example by keyboard or mouse events). Backed by inactivity timeout.	Each user-triggered event extends the current interval (or creates a new one, if there is none). Once the inactivity threshold is reached or the event stream ends, we close the current interval.
`EclipseActive` † *	Interval in which the IDE had the focus on the computer.	FeedBaG++ monitors the active window in the same way as WatchDog does. We group events into intervals.
`Perspective`	Interval describing which perspective the IDE was in (Debugging, regular Java development, ...).	We approximate the manually changed Eclipse `Perspectives`, with Visual Studio's automatically changing perspectives.
`WatchDogView` *	Interval that is created when the user consults the immediate WatchDog statistics. Only available in the Eclipse IDE.	*Not provided in FeedBaG++.*
`EclipseOpen` †	Interval in which the IDE was open. If the computer is suspended, the `EclipseOpen` is closed and the current sessions ends. Upon resuming, a new `EclipseOpen` interval is started, discarding the time in which the computer was sleeping. Each session has a random, unique identifier.	FeedBaG++ generates specific events that describe the IDE state. From the start-up and shutdown events of the IDE, we generate `EclipseOpen` intervals.

† As of WatchDog 1.5, we support multiple IDEs, so better interval names would have been `TestExecution`, `IDEActive`, and `IDEOpen`.
* Not shown in Figure 4.5.

4

file X (`Reading`) to typing in file Y (`Writing`), we immediately end the currently opened interval. WatchDog closes all such activity-based intervals after an inactivity timeout of 16 seconds, so that we adjust for breaks and interruptions. A timeout length of roughly 15 seconds is standard in IDE-based observational plugins [65, 192, 193]. Most interval types may overlap. For example, WatchDog always wraps `Typing` or `Reading` intervals inside a `UserActive` interval (which it, in turn, wraps in an `EclipseActive`, `Perspective`, and `EclipseOpen` interval). However, `Reading` and `Typing` intervals are by nature mutually exclusive. We refer to an IDE session as the time span in which the IDE was continuously running (even in the background) and not closed or interrupted, for example, because the developer suspended the computer. All intervals that belong to one IDE session are hence wrapped within one `EclipseOpen` interval, ① in Figure 4.5.

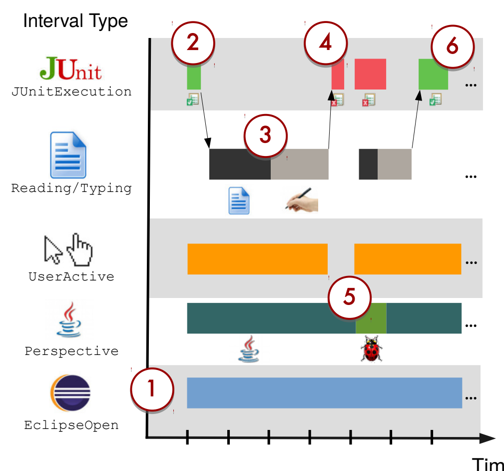


Figure 4.5: Exemplary workflow visualization with intervals. Table 4.1 describes the interval types in the same order as they appear in the different rows.

We enrich `Reading` and `Typing` intervals with different information about the underlying file. To all intervals we add a hash of the filename and its file type, such as XML or Java class. For Java classes, we add their SLOC and classify them as production or test code. As our churn measure for the size of a change, we also add the Levenshtein edit distance [194] between the content of the file before and after the modification during the interval to `Typing` intervals.

Test Recognition. WatchDog has four different recognition categories for test classes (see Table 4.1): To designate the file as a test that can be executed in the IDE, we require the presence of at least one JUnit `import` together with at least one method that has the `@Test` annotation or that follows the `testMethod` naming convention. This way, we support both JUnit3 and JUnit4. Furthermore, we recognize imports of common Java test frameworks and their annotations (Mockito, PowerMock). As a last resort, we recognize when a file contains "Test" in its file name or the project file path. It is a common convention to pre- or postfix the names of test files with `Test` [179], or to place all test code in one sub-folder. For example, the standard Maven directory layout mandates that tests be placed under `src/test/java` [195]. Thereby, we can identify and differentiate between all tests that employ standard Java testing frameworks as test runners for their unit, integration, or system tests, test-related utility classes, and even tests that are not executable in the IDE. We consider any Java class that is not a test according to this broad test recognition strategy to be production code.

FeedBaG++-to-WatchDog Interval Transformation

In contrast to the native WatchDog clients, FeedBaG++ provides us with a raw event stream (see Section 4.1.1). To feed FeedBaG++ data into the WatchDog pipeline, we

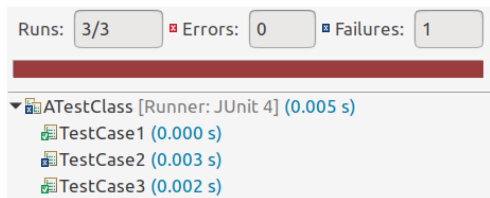


Figure 4.6: Eclipse's visualization of the `JUnitExecution` constituents.

derive intervals via a post factum analysis of FeedBaG++ data. In addition to this technical difference, several minor semantic differences exist in the instrumented IDEs. We had to find congruent concepts for them and transform FeedBaG++ events to intervals.

Concept Mapping. The Eclipse, IntelliJ, and the Visual Studio IDEs are similar conceptually, yet differ in some implementation details important to our study. In addition to IDE concepts, we had to map C# concepts to their Java counterparts.

One such central difference is the different testing frameworks available in the C# ecosystem. FeedBaG++ recognizes the same four categories of test classes described in Section 4.1.3: To designate a file as a test that can be executed in Visual Studio, we require an import of one of the .NET testing frameworks NUnit, XUnit, MSUnit, csUnit, MbUnit, or PetaTest. Furthermore, we recognize imports of the C# mocking frameworks moq, Rhino.Mocks, NSubstitute, and Simple.Mocking.

A difference between Visual Studio and Eclipse is that the former does not have perspectives that developers can manually open, but instead it automatically switches between its *design view* for writing code, and its *debug view* for debugging a program run. We map the concept of these Visual Studio views to the `Perspective` intervals in WatchDog.

Arguably the largest difference between IDEs is how they manage different projects and repositories. Eclipse organizes source code in a *workspace* that may contain many potentially unrelated *projects*. IntelliJ groups several *modules* in a *project*. Visual Studio organizes code in a *solution*, which contains a number of usually cohesive *projects*. In Java, a single project or module often contains both the production code and test code. This is not the case in Visual Studio, where the two kinds of source code are typically split into two separate projects. If not accounted for, this leads to a higher number of observed projects in Visual Studio and distorts the answers to some of our project-level research questions. To counter this problem, we need functions to map test code from one project to its corresponding production code in another. The notion of a Visual Studio *solution* and even more so, IntelliJ's *project* matches the definition of a *Watchdog project*, understood as a cohesive *software development effort*. To avoid confusion about the overloaded "project" term, we asked the user explicitly whether "all Eclipse projects in this workspace belong to one 'larger' project?" in the WatchDog registration dialogues (see Section 4.1.2).

FeedBaG++ does not measure the Levenshtein distance in `Typing` intervals. However, WatchDog data shows that the edit distance generally correlates strongly with the number of edits: The number of production code edits correlates at $\rho = 0.88$ with production code churn, i.e., the amount of changed code [196], and the number of test edits is correlated at $\rho = 0.86$ with test code churn. Hence, we use the number of edits as a proxy for

the missing churn in FeedBaG++ data.

Event Transformation. As a second step, we transformed the event stream to intervals. We re-implemented transformation rules that work on the raw FeedBaG++ event stream based on the interval detection logic that the WatchDog plugin family performs within the IDE. We then store it in WatchDog's central NoSQL database store (see Figure 4.1). In the right column of Table 4.1, we sketch how we derive the various WatchDog interval types from the events that FeedBaG++ captures. From there, we simply re-use the existing WatchDog analysis pipeline.

4.2 Research Methods

In this section, we describe the methods with which we analyze the data for our research questions.

4.2.1 Correlation Analyses (RQ III.1, RQ III.2)

We address our research questions RQ III.1 and RQ III.2 with the help of correlation analyses. For example, one of the steps to answer RQ III.1 is to correlate the test code churn introduced in all `Typing` intervals with the number of test executions.

Intuitively, we have the assumption that if developers change a lot of code, they would run their tests more often. Like all correlation analyses, we first compute the churn and the number of test executions for each IDE session and then calculate the correlation over these summed-up values of each session. IDE sessions form a natural divider between work tasks, as we expect that developers typically do not close their IDE or laptop at random, but exactly when they have finished a certain task or work step (see Table 4.1).

4.2.2 Analysis of Induced Test Failures (RQ III.3)

Algorithm 1 Sketch of Test Failure Percentage Calculation

```
 1: procedure calcFailingTestPercentage(project)
 2:     tcs.ok ⟵ successful(testcases(project))        ▷ List of every single successful execution of a test case
 3:     tcs.failed ⟵ failed(testcases(project))    ▷ List of every single failed or errored execution of a test case
 4:     tcs ⟵ tcs.ok ∪ tcs.failed
 5:     if n(unique(tcs) < 10) then                                              ▷ Not enough test cases
 6:         return
 7:     end if
 8:     fail.tc     ▷ Map between a test case name (key) and the relative amount of test executions in which it
    failed (value)
 9:     for tc ∈ unique(tcs.failed) do
10:         fail.tc(tc) ⟵ n(tc ∈ tcs) / n(failed(tests(project)))
11:     end for
12:     values(fail.tc) ⟵ order(values(fail.tc), descending)
13:     fail.perc  ▷ Per percentage of all test cases, returns which percentage of failures they are responsible for
    ▷ Invariants: fail.perc(0) = 0 and fail.perc(1) = 1
14:     for i ∈ {0%, 0.1%, 0.2%, ..., 100%} do
15:         first.i.tcs ⟵ head(fail.rate, round(i · n(unique(tcs))))
16:         failure.rate(i) ⟵ sum(values(first.i.tcs))
17:     end for
18:     return fail.perc
19: end procedure
```

We abstract and aggregate the tests of multiple projects to derive general statements like "only 25% of tests are responsible for 75% of test failures in the IDE." Algorithm 1 outlines the steps we use to count the number of executed test cases and the number of corresponding test failures they have caused per project. We iterate over all failed test cases (line 9), determine which percentage of failed test executions they are responsible for (line 10) and put the resulting list of test cases in descending order, starting with the test case with the highest responsibility of test failures (line 12). We then normalize the absolute count numbers to the relative amount of test cases in the project (line 14) by calling calcFailingTestPercentage on every project, average the results so that each project has the same weight in the graph, and plot them.

The algorithm makes assumptions that lead to a likely underestimation of the percentage of test failures caused by a specific test: First, it assumes that test names are stable. If test names change during our field study, they count as two different tests, even though their implementation might stay the same. Second, it excludes projects that only have a small number of test cases (< 10). If, for instance, a project only has two test cases, the result that 50% (i.e., one) of them is responsible for all test failures would be too coarse-grained for our purposes.

4.2.3 Sequentialization of Intervals (RQ III.3, RQ III.4)

For RQ III.3 and RQ III.4, we need a linearized stream of intervals following each other. We generate such a sequence by ordering the intervals according to their start time. For example, in Figure 4.5, the sequenced stream after the first test failure in (4) is:
`Failing Test` ⟶ `Switch Perspective` ⟶ `Start JUnit Test` ⟶ `Read Production Code` ⟶ ...

4.2.4 Test Flakiness Detection (RQ III.3)

Flaky tests are defined as tests that show non-deterministic runtime behavior: they pass one time and fail another time without modifications of the underlying source code or test [197]. Applied to the WatchDog interval concept, we look for subsequent executions of test cases embedded in `JUnitExecution` intervals that have no `Typing` interval to either production or source code in-between them in the above linearized interval stream from Section 4.2.3. If the result of those subsequent executions differs, for example `Failing Test` ⟶ ... ⟶ `Passing Test`, we regard such a test as flaky. To control for external influences, we only do this within the confines of a session, not across sessions. Otherwise, the risk for external influences becomes too large, for example through updating the project via the command line without our IDE plugin noticing.

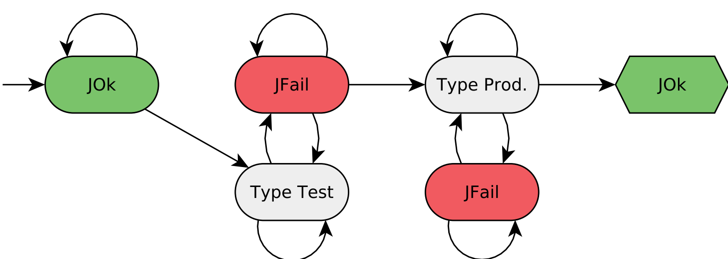
4.2.5 Recognition of Test-Driven Development (RQ III.4)

Test-Driven development (TDD) is a software development process originally proposed by Beck [5]. While a plethora of studies have been performed to quantify the supposed benefits of TDD [198, 199], it is unclear how many developers use it in practice. In RQ III.4, we investigate how many developers follow TDD to which extent. In the following, we apply Beck's definition of TDD to the WatchDog interval concept, providing a *formally*

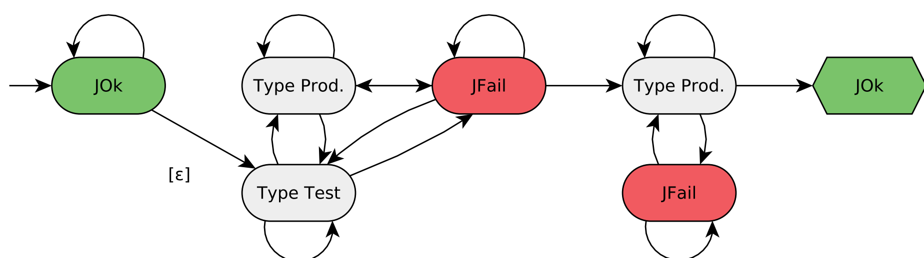
verifiable definition of TDD in practice. Since TDD is a process sequence of connected activities, it lends itself toward modeling as a state machine [200].

TDD is a cyclic process comprising a functionality-evolution phase depicted in Figure 4.7, optionally followed by a functionality-preserving refactoring phase depicted in Figure 4.8. We can best illustrate the first phase with the strict non-finite automaton (NFA, [201]) in Figure 4.7a and our developer Wendy, who is now following TDD: before Wendy introduces a new feature or performs a bug fix, she assures herself that the test for the production class she needs to change passes (`JOk` in Figure 4.7 stands for a `JUnitExecution` that contains a successful execution of the test under investigation). Thereafter, she first changes the test class (hence the name "test-first" software development) to assert the precise expected behavior of the new feature or to document the bug she is about to fix. We record such changes in a `Typing` interval on test code. Naturally, as Wendy has not yet touched the production code, the test must fail (`JFail`). Once work on the test is finished, Wendy switches to production code (`Type Prod.`), in which she makes precisely the minimal required set of changes for his failing test to pass again (`JOk`). The TDD cycle can begin anew.

When we applied this strict TDD process, we found that it is difficult to follow in reality, specifically the clear separation between changes to test code and later changes to production code. Especially when developing a new feature like the `Board` of a board game in Figure 4.9, developers face compilation errors during the test creation phase of



(a) Strict



(b) Lenient

Figure 4.7: Strict and lenient NFAs of TDD. `JOk` stands for a passing and `JFail` for a failing test execution (`JUnitExecution`).

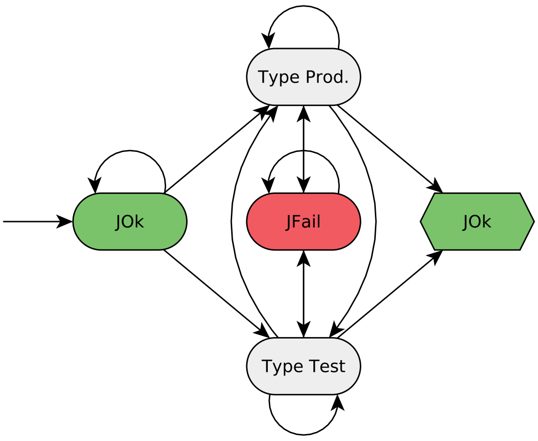


Figure 4.8: NFA for the refactoring phase of TDD.

```
class BoardTest {
    @Test
    void verifyHeight() {
        Board board = new Board();
        assertThat(board.getHeight()).isEqualTo(3);
    }   Board cannot be resolved to a type
```

Figure 4.9: Compile errors while creating a TDD test.

TDD, because the class or method they want to assert on (`Board`) does not exist yet, since the test has to be created before the production code. To be able to have an executing, but failing test, they have to mix in the modification or creation of production code. Moreover, developers often know the result of a test without executing it (for example, because it contains obvious compile errors like in Figure 4.9), and that a test case succeeds before they start to work on it (for example, because they fixed the test on their previous day at work). To adjust for these deviations between a strict interpretation of TDD and its application, we created the lenient non-finite automaton (ϵ-NFA, [201]) in Figure 4.7b, which is more suitable for the recognition of TDD in practice. Due to the ϵ-edge, a TDD cycle can directly start with modifications of test code.

TDD does not only comprise a functionality-changing phase, but also the code refactor phase depicted in Figure 4.8. In this phase, developers have the chance to perform functionality-preserving refactorings. Once they are finished with refactoring, the tests must still pass [5]. It is impossible to separate changes between production and test classes in the refactoring phase in practice, as the latter rely on the API of the first.

To assess how strictly developers follow TDD, we convert all three NFAs to their equivalent regular expressions and match them against the linearized sequence of intervals (see Section 4.2.3). For a more efficient analysis, we can remove all intervals from the sequentialized stream except for `JUnitExecution` and `Typing` intervals, which we need to recog-

Table 4.2: Descriptive statistics of study data and participants.

IDE	Language	Plugin & Version	#Users	#Countries	#Projects	Work Time	#Sessions	#Intervals	Collection Period	Runtime
EC	Java	WD 1.0 – 2.0.2	2,200	115	2,695	146.2 years	66,623	12,728,351	15 Sept. 2014 – 1 March 2017	488 min
IJ	Java	WD 1.5 – 2.0.2	117	30	212	3.9 years	5,511	950,998	27 June 2015 – 1 March 2017	25 min
AS	Java	WD 1.7 – 2.0.2	71	27	178	1.0 year	2,717	347,468	26 Jan. 2016 – 1 March 2017	13 min
VS	C#	FB 0.1010 – 0.1015	55	$\ll 55$	423	9.7 years	2,259	239,866	12 June 2016 – 1 March 2017	13 min
Σ	Java, C#	WD, FB	2,443	118	3,508	161 years	77,110	14,266,683	15 Sep. 2014 – 1 March 2017	541 min
Σ_{CN}	Java, C#	WD, FB	181	38	434	33.9 years	15,928	3,137,761	15 Sep. 2014 – 1 March 2017	83 min

nize TDD. To be able to draw a fine-grained picture of developers' TDD habits, we performed the analysis for each session individually. We count refactoring activity towards the total usage of TDD. The portion of matches in the whole string sequence gives us a precise indication of a developer's adherence to TDD.

4

4.2.6 Statistical Evaluation (RQ III.1–RQ III.5)

When applying statistical tests in the remainder of this chapter, we regard results as significant at a 95% confidence interval ($\alpha = 0.05$), i.e., iff $p \leqslant \alpha$. All results of tests t_i are statistically significant at this level, i.e., $\forall i : p(t_i) \leqslant \alpha$.

For each test t_i, we first perform a *Shapiro-Wilk Normality test* s_i [202]. Since all our distributions significantly deviate from a normal distribution according to Shapiro-Wilk ($\forall i : p(s_i) < 0.01 \leqslant \alpha$), we use non-parametric tests: 1) For testing whether there is a significant statistical difference between two distributions, we use the non-parametric *Wilcoxon Rank Sum test.* 2) For performing correlation analyses, we use the non-parametric *Spearman rank-order (ρ) correlation coefficient* [203]. Hopkins's guidelines facilitate the interpretation of ρ [53]: they describe $0 \leqslant |\rho| < 0.3$ as no, $0.3 \leqslant |\rho| < 0.5$ as a weak, $0.5 \leqslant |\rho| < 0.7$ as a moderate, and $0.7 \leqslant |\rho| \leqslant 1$ as a strong correlation.

4.3 Study Participants

In this section, we first explain how we attracted study participants, report on their demographics, and then show how we produced a normalized sample.

4.3.1 Acquisition of Participants

We reached out to potential developers to install WatchDog (WD) and FeedBaG++ (FB) in their IDE by:

1. Providing project websites (WD, FB).[4]
2. Raffling off prizes (WD).
3. Delivering value to WatchDog users in that it gives feedback on their development behavior (WD).
4. Writing articles in magazines and blogs relevant to Java and Eclipse developers: Eclipse Magazin, Jaxenter, EclipsePlanet, Heise News (WD).
5. Giving talks and presentations at developer conferences: Dutch Testing Day, EclipseCon (WD).

[4] http://www.testroots.org, http://kave.cc

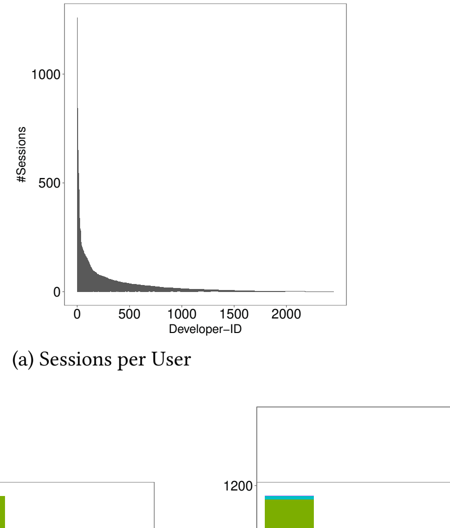
6. Presenting at research conferences [65, 66, 184, 192, 204] (WD, FB).
7. Participating in a YouTube Java Developer series [205] (WD).
8. Penetrating social media: Reddit, Hackernews, Twitter, Facebook (WD, FB).
9. Approaching software development companies (WD, FB).
10. Contacting developers, among them 16,058 Java developers on GitHub (WD).
11. Promoting our plugins in well-established Eclipse [206], IntelliJ [207], and Visual Studio [208] marketplaces (WD, FB).
12. Launching a second marketplace that increases the visibility of scientific plugins within the Eclipse ecosystem, together with the Eclipse Code Recommenders project [209] (WD).
13. Promoting the plugin in software engineering labs at TU Darmstadt (FB).
14. Approaching an electrical engineering research group working with Visual Studio (FB).

We put emphasis on the testing reports of WatchDog to attract developers interested in testing. Instead, for FeedBaG++, we mainly advertised its integrated code completion support.

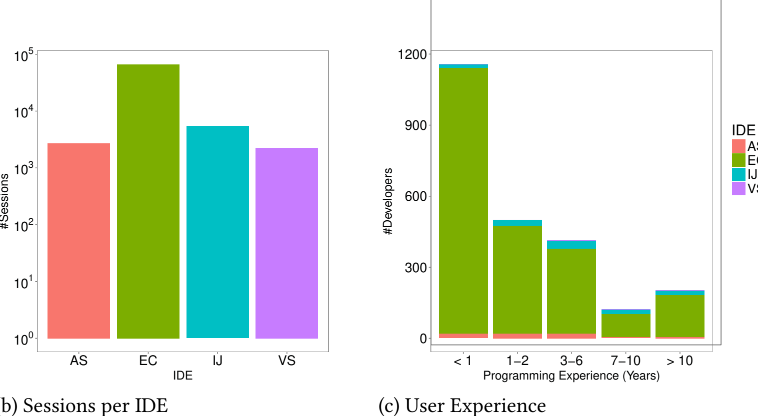
4.3.2 Demographics of Study Subjects

Table 4.2 and Figure 4.10 provide an overview of the observational data we collected for this chapter. In total, we observed 14,266,683 user interactions (so-called intervals, see Section 4.1.1) in 77,110 distinct IDE sessions. Figure 4.10a shows how 10% of our 2,443 users contributed the wealth of our data (80%). The majority of users and, thus, data stems from the Eclipse IDE, shown in Figure 4.10b. Reasons include that the collection period for Eclipse is longer than that of the other IDEs and that we advertised it more heavily. In this chapter, we report on an observatory field study stretching over a period of 2.5 years, on data we collected from the 15th of September 2014 to March 1st 2017, excluding student data that we had analyzed separately [65], but including our original developer data [66]. Data periods for other plugins are shorter due to their later release date. As we updated WatchDog to fix bugs and integrate new features (see Section 4.1.1), we also filtered out data from deprecated versions 1.0 and 1.1.

Our users stem from 118 different countries. The most frequent country of is the United States (19% of users), followed by China (10%), India (9%), Germany (6%), The Netherlands (4%), and Brazil (4%). The other half comes from the 112 remaining countries, with less than 4% total share each. Our developers predominately use some variant of Windows (81% of users), MacOS (11%), or Linux (8%). Their programming experience in Figure 4.10c is normally distributed (a Shapiro-Wilks test fails to reject the null hypothesis that it is not normally distributed at $p = 0.15$). Generally, we have more inexperienced (< 3 years, 69% of users) than experienced users. On the other hand, very experienced developers (≥ 7 years) represent more than 13% of our population.



(a) Sessions per User



(b) Sessions per IDE (c) User Experience

Figure 4.10: Distributions of the number of sessions per developer (all IDEs), per IDE (log scale), and their programming experience (WatchDog only).

Overall, the 2,443 participants registered 3,508 unique projects. The registered projects stem from industry as well as famous open-source initiatives, such as the Apache Foundation, but also include private projects.

Using the average work time for OECD countries of 1770 hours per year,[5] we observed a total work time of 161 developer years on these registered projects in the IDE. The last column in Table 4.2 denotes the runtime of our analysis pipeline running on a dedicated server with 128GB RAM using eight Intel Xeon E5-2643 cores at 3.50GHz.

This chapter broadens our single-IDE study on developer testing in the IDE to a very large set of developers (a ten-fold increase over our original WatchDog data [65]). Survey responses from 2,291 registrations of WatchDog users and projects complement our

[5] http://stats.oecd.org/index.aspx?DataSetCode=ANHRS

technical IDE observations that now stem from four IDEs in two mainstream programming languages. FeedBaG++ data stems from the March 1st, 2017 event data set [210].

4.3.3 Data Normalization

As discussed in Section 4.3.2, the majority of our intervals (80%) stems from only 378 users. The long tail of users that contributed only little data might impact some of our analyses (see Figure 4.10a). Conversely, the large amount of data we received from few developers might affect our results with a bias toward the individual development preferences of those few developers. To reduce both biases, we cap and normalize our data using stratified random sampling on the number of sessions per user. We chose sessions, because they are at a finer granularity than projects, but still allow analyses such as the TDD recognition, which would not work when sampling random intervals that have no connection to each other.

We first order our users by the number of sessions each user submitted and cap at below the user at which we reached 80% of all sessions. This leaves in users with at least 88 sessions each, effectively removing the bulk of users who barely contributed data and might, thus, skew user- or project-based analyses. The problem that few users have a disproportionately large impact on the analyzed data remains. Hence, we normalize the data by randomly sampling 88 of the available sessions for each user. After this, every user has the same influence on the results in our new capped, normalized data set, depicted as Σ_{CN} in Table 4.2. In comparison to our overall population Σ, the distribution of originating countries and IDEs is similar. The only apparent change in population demographics is an almost three-fold increase of very experienced developers to 32% in Σ_{CN}.

Since our study is a large-scale observatory field study, we primarily use our non-normalized data set Σ when answering research questions. Filtering criteria remain to some extent arbitrary and might induce a bias themselves. Whenever there is a significant difference in the capped normalized data set Σ_{CN}, we report and discuss this in the answer to the appropriate research question.

4.4 Results

In the following, we report the results to each of our research questions individually per subsection.

4.4.1 RQ III.1: Which Testing Patterns Are Common In the IDE?

To answer how and why developers test, we must first assess:

RQ III.1.1 How Common Is Codified Testing in the IDE?

When we apply our broad recognition of test classes as described in Section 4.1.3 and Table 4.1, we detect test activities in only 43% of projects in our data set (EC: 46%, IJ: 26%, AS: 28%, VS: 26%), meaning that, in total, only 1,498 projects out of 3,508 contain tests that a user either read, changed, or executed in the IDE. This is one of the analyses that is potentially impacted by data skews due to a short amount of observed development behavior for many users. However, even in Σ_{CN}, only 255 projects out of 434 (58%) showed testing activity.

If we restrict the recognition to tests that can be run through the IDEs, we find that 594 projects have such tests (EC: 436, IJ: 88, AS: 27, VS: 40), about 17% of the registered projects (EC: 16%, IJ: 22%, AS: 15%, VS: 9%). In Σ_{CN}, this percentage is somewhat higher at 29%, with 124 projects with executable tests. By comparing the WatchDog projects IDE data to what developers claimed in the survey, we could *technically* detect JUnit tests in our interval data (as either `Reading`, `Typing`, or `JUnitExecution`) for only 43% of projects that should have such tests according to the survey (EC: 42%, IJ: 61%, AS: 32%). Here, we find the only obvious difference in Σ_{CN}, where the percentage of users who claimed to have JUnit tests and who actually had them, is 73%.

Our second sub-research question is:

RQ III.1.2 How Frequently Do Developers Execute Tests?

Of the 594 projects with tests, we observed in-IDE test executions in 431 projects (73%, EC: 75%, IJ: 68%, AS: 37%, VS: 80%). In these 431 projects, developers performed 70,951 test runs (EC: 63,912, IJ: 3,614, AS: 472, VS: 2,942). From 59,198 sessions in which tests could have been run because we observed the corresponding project to contain an executable test at some point in our field study, we observed that in only 8% or 4,726 sessions (EC: 8.1%, IJ: 7.4%, AS: 3.4%, VS: 8.9%) developers made use of them and executed at least one test. The average number of executed tests per session is, thus, relatively small, at 1.20 for these 431 projects. When we consider only sessions in which at least one test was run, the average number of test runs per session is 15 (EC: 15.3, IJ: 11.1, AS: 7.6, VS: 17.9).

When developers work on tests, we expect that the more they change their tests, the more they run their tests to inform themselves about the current execution status of the test they are working on. RQ III.1.3 and following can, therefore, give an indication as to why and when developers test:

RQ III.1.3 Do Developers Execute Their Test Code Changes?

The correlation between test code changes and the number of test runs yields a moderately strong $\rho = 0.65$ (EC: 0.64, IJ: 0.60, AS: 0.41, VS: 0.66) in our data sample (p-value < 0.01). In other words, the more changes developers make to a test, the more likely are they to execute this test (and vice versa).

A logical next step is to assess whether developers run tests when they change the production code: Do developers assert that their production code still passes the tests?

RQ III.1.4 Do Developers Test Their Production Code Changes?

The correlation between the number of test runs and number of production code changes is generally weaker, with $\rho = 0.39$ (EC: 0.38, IJ: 0.47, AS: 0.20, VS: 0.60) and p-value < 0.01.

Finally, in how many cases do developers modify their tests, when they touch their production code (or vice versa), expressed in:

RQ III.1.5 Do Developers Co-Evolve Test and Production Code?

In this case, the Spearman rank correlation test indicates no correlation ($\rho = 0.31$, EC: 0.26, IJ: 0.58, AS: 0.43, VS: 0.73) between the number of changes applied to test and production code. This means that developers do not modify their tests for every production code change, and vice versa.

Table 4.3: Descriptive statistics for RQ III.2 and RQ III.3 in the $\sum$ data (similar across IDEs, hence abbreviated).

Variable	Unit	Min	25%	Median	Mean	75%	Max	Log-Histogram
`JUnitExecution` duration	Sec	0	0	0.5	107.2	3.1	652,600	
Tests per `JUnitExecution`	Items	1	1	1	5.0	1	2,260	
Time to fix failing test	Min	0	0.9	3.7	44.6	14.9	7,048	
Test flakiness per project	Percent	0	0	0	12.2	15.8	100	

4.4.2 RQ III.2: What Characterizes The Tests Developers Run In The IDE?

When developers run tests in the IDE, they naturally want to see their execution result as fast as possible. To be able to explain how and why developers execute tests, we must, therefore, first know how long developers have to wait before they see a test run finish:

RQ III.2.1 How Long Does a Test Run Take?

In all IDEs except for Visual Studio, 50% of all test executions finish within half a second (EC: 0.42, AS: 1.8s, IJ: 0.47s, VS: 10.9s), and over 75% within five seconds (EC: 2.37s, IJ: 2.17s, AS: 3.95s, VS: 163s), see Table 4.3 for the average values. Test durations longer than one minute represent only 8.4% (EC: 4.2%, IJ: 6.9%, AS: 6.1%, VS: 32.0%) of the `JUnitExecutions`.

Having observed that most test runs are short, our next step is to examine whether short tests facilitate testing:

RQ III.2.2 Do Quick Tests Lead to More Test Executions?

To answer this research question, we collect and average the test runtime and the number of times developers executed tests in each session, as in Section 4.4.1. Then, we compute the correlation between the two distributions. If our hypothesis was true, we would receive a negative correlation between the test runtime and the number of test executions. This would mean that short tests are related to more frequent executions. However, the Spearman rank correlation test shows that this is not the case, as there is no correlation at $\rho = 0.27$ (EC: 0.40, IJ: 0.24, AS: 0.83, VS: 0.41). In Android Studio's case, the opposite is true, indicating a strong relationship between the runtime of a test and its execution frequency. Combined with the fact that only a small number of tests are executed, our results suggest that developers explicitly select test cases [211]. While test selection is a complex problem on build servers, it is interesting to investigate how developers perform it locally in their IDE:

RQ III.2.3 Do Developers Practice Test Selection?

A test execution that we capture in a `JUnitExecution` interval may comprise multiple child test cases. However, 86% of test executions contain only one test case (EC: 86%, IJ: 88%, AS: 80%, VS: 85%), while only 7.7% of test executions comprise more than 5 tests (EC: 7.8%, IJ: 4.8%, AS: 7.6%, VS: 10.3%), and only 2.2% more than 50 tests (Table 4.3, EC: 2.2%, IJ: 0.1%, AS: 0.0%, VS: 4.4%).

Test selection likely happened if the number of executed tests in one `JUnitExecution` is smaller than the total number of tests for the given project (modulo test renames, moves, and deletions). The ratio between these two measures allows us to estimate the percentage of selected test cases. If it is significantly smaller than 100%, developers practiced test

selection. Our data in Table 4.3 shows that 86.4% of test executions include only one test case.

To explain how and why this test selection happens with regard to a previous test run, we investigate two possible scenarios: First, we assume that the developer picks out only one of the tests run in the previous test execution, for example to examine why the selected test failed. In the second scenario, we assume that the developer excludes a few disturbing tests from the previous test execution. In the 1719 cases in which developers performed test selection, we can attribute 94.6% (EC: 94.6%, IJ: 91.8%, AS: 82.4%, VS: 95.5%) of selections to scenario 1, and 4.9% (EC: 5.2%, IJ: 0.0%, AS: 5.8%, VS: 3.6%) to scenario 2. Hence, our two scenarios together are able to explain 99.5% (EC: 99.8%, IJ: 91.8%, AS: 88.2%, VS: 99.1%) of test selections in the IDE.

4.4.3 RQ III.3: How Do Developers Manage Failing Tests?

Having established how often programmers execute tests in their IDE in the previous research questions, it remains to assess:

RQ III.3.1 How Frequently Do Tests Pass and Fail?

There are three scenarios under which a test execution can return an unsuccessful result: The compiler might detect compilation errors, an unhandled runtime exception is thrown during the test case execution, or a test assertion is not met. In either case, the test acceptance criterion is never reached, and we therefore consider them as a test failure, following JUnit's definition.

In the aggregated results of all observed 70,951 test executions, 57.4% of executions fail, i.e., 40,700 `JUnitExecutions` (EC: 57.4%, IJ: 60.7%, AS: 56.8%, VS: 43.2%), and only 42.6% pass successfully. Moreover, when we regard the child test cases that are responsible for causing a failed test execution, we find that in 86% (EC: 95%, IJ: 84%, AS: 88%, VS: 94%) of test executions only one single test case fails, and is, thus, responsible for making the whole test execution fail, even though other test cases from the same test class might pass, as exemplified in Figure 4.6.

To zoom into the phenomenon of broken tests, we ask:

RQ III.3.2 Are All Test Cases Equally Responsible for Test Failures?

In this question, we regard all test cases that have ever been executed and observed. We then calculate and track how many times each of them failed, as described in detail in Section 4.2.2. Since we cannot track renames of files and, therefore, treat them as two different files, it is likely that the real error percentage for test cases is slightly higher. Figure 4.11 depicts the results, showing that only 25% of test cases are responsible for over 75% of test failures in Eclipse and Visual Studio. In all IDEs, 50% of test cases are responsible for over 80% of all test errors. While slightly lower for IntelliJ-based IDEs, the failure and growth rate of the curve is similar across IDEs, suggesting a near-logarithmic growth.

As developers apparently often face test failures, we ask:

RQ III.3.3 How Do Developers React to a Failing Test?

For each failing test execution in our data sets, we generate a linearized stream of subsequently following intervals, as explained in Section 4.2.3. By counting and summing up developers' actions after each failing test for up to 3.3 minutes (200 seconds), we can draw a precise picture of how developers manage a failing test in Figure 4.12. Across all

IDEs, the most widespread immediate reaction in ~50% of cases within the first seconds is to read test code.[6] The second most common reaction, at stable 20% of reactions across the time, is to read production code.

The next most common reactions – switching focus away from the IDE (for example, to turn to the web browser), switching perspective in the IDE (for example to a dedicated debugging perspective), typing test code, and being inactive – appear in different order among IDEs. Typing test code, however, is a more common reaction to a failing test in all IDEs than typing production code. Starting another test execution is a fairly common course of action within the first minute across all IDEs, reaching ~15% frequency. Switching perspective is only prevalent in the first seconds (see Figure 4.12d), since it is an automated feature of Visual Studio (see Section 4.1.3). Altogether quitting the IDE almost never happens and is, therefore, not shown. After two minutes (120 seconds), the

[6] While writing this extension, we uncovered a bug in the analysis code to RQ III.3.3. The bug swapped the "Read Test Code" with the "Read Production Code" label. This lead us to wrongly claim in the original WatchDog paper [66] that developers dived into offending production code first, which was never the case.

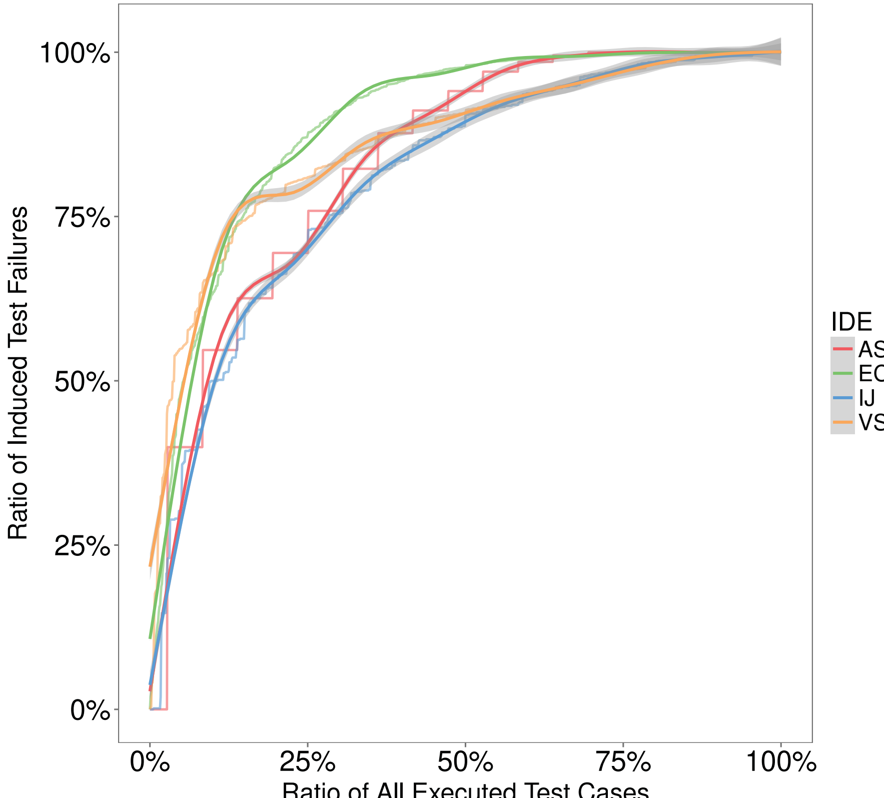
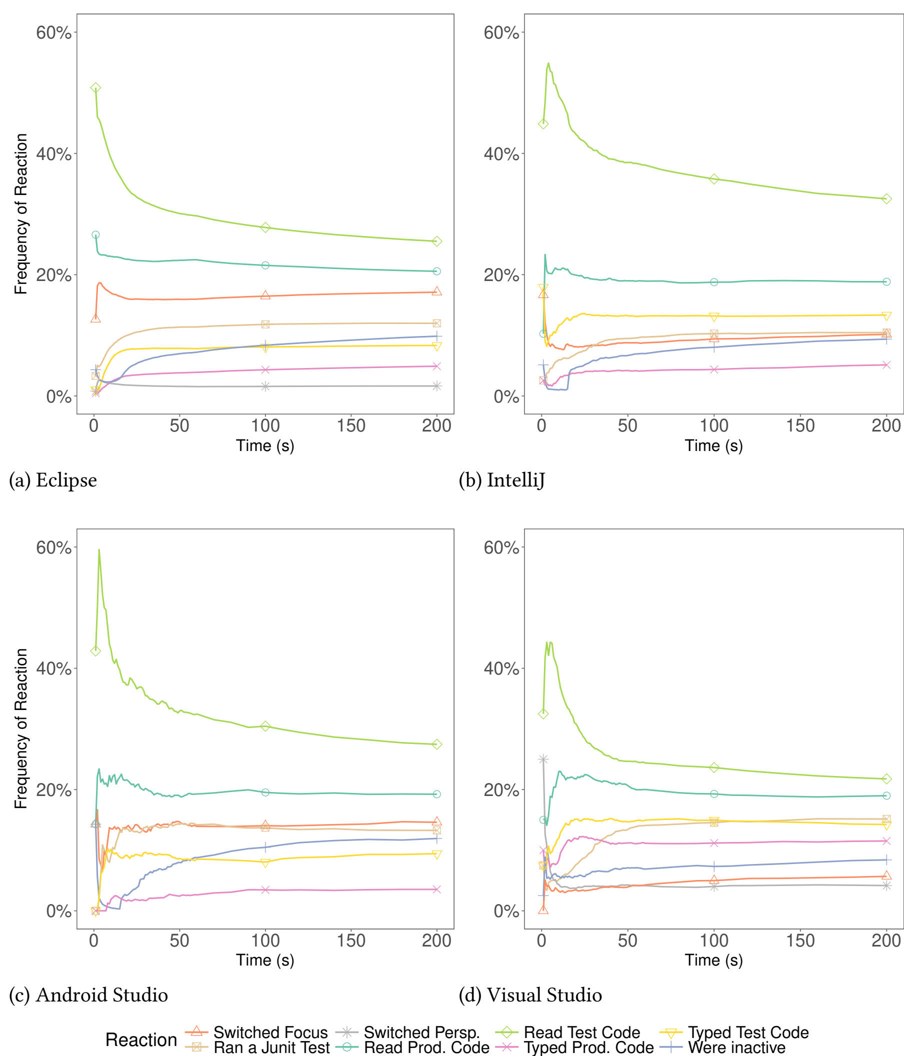


Figure 4.11: Accumulated error responsibility of test cases per IDE. Based on 134 projects with ≥ 10 run test cases (EC: 112, IJ: 9, AS: 1, VS 12).



(a) Eclipse (b) IntelliJ

(c) Android Studio (d) Visual Studio

Figure 4.12: Frequency of immediate reactions to a failing test over time, separated by IDE.

reactions trend asymptotically toward their overall distribution, with little variability.

The logical follow-up to RQ III.3.3 is to ask whether developers' reactions to a failing test are in the end successful, and:

RQ III.3.4 How Long Does It Take to Fix a Failing Test?

To answer this question, we determine the set of unique test cases per project and their execution result. The 40,700 failing test executions were caused by 15,696 unique test classes according to their file name hash (EC: 13,371, IJ: 959, AS: 94, VS: 1,271). We never saw a successful execution of 32% (EC: 28%, IJ: 50%, AS: 46%, VS: 54%) of tests, and at least

one successful execution of the others.

For the 10,701 failing tests that we know have been fixed later, we examine how long developers take to fix a failing test. Table 4.3 shows that a quarter of test repairs happen within less than a minute, half within 4 minutes, and 75% within 15 minutes.

One reason why in some cases the time between a failing and succeeding test might be so short is that developers did not actually have to make repairs to their tests. Instead, they might have just executed the tests without changes, since it might be flaky. A flaky test is a test that shows non-deterministic pass behavior [212, 213], meaning it (randomly) fails or succeeds. To answer this question, we ask for the IDE:

RQ III.3.5 Do Developers Experience Flaky Tests?

Following the research method described in Section 4.2.4, we measure the "test flakiness" per project, the percentage of tests that show non-deterministic behavior despite the fact that there are no changes to the project in the meantime, including changes to test, production, or configuration files. Table 4.3 shows that the mean flakiness value is 12.2%, with outliers of zero and 100% flaky test percentages.

4.4.4 RQ III.4: Do Developers Follow TDD In The IDE?

In RQ III.4, we aim to give an answer to the adoption of TDD in practice.

Our results reveal that sessions of only 43 developers match against a strict TDD definition, the top NFA in Figure 4.7a (EC: 42, IJ: 0, AS: 0, VS: 1). This makes 1.7% of all developers, or 11.8% of developers who executed tests, see Section 4.4.1. In total, only 2.2% of sessions with test executions contain strict TDD patterns. Only one developer uses strict TDD in more than 20% of the development process on average. Seven of the 43 developers use TDD for at least 5% of their development. The remaining 35 developers use strict TDD in less than 5% of their intervals. Refactoring is the dominant phase in TDD: 39 of the 43 developers did some form of refactoring. At 69%, the majority of the intervals of the 43 developers are devoted to the refactoring phase of TDD (depicted in Figure 4.8). Most developers who practiced strict TDD have a long programming experience: 23 declared an experience between 7 and 10 years.

Sessions from 136 developers match against the lenient TDD NFA in Figure 4.7b (EC: 42, IJ: 18, AS: 3, VS: 3). This makes 5.6% of all developers, or 37% of developers who executed tests (EC: 15%, IJ: 38%, AS: 33%, VS: 19%), see Section 4.4.1. Sixteen developers use lenient TDD in more than 20% of their intervals, including the developer who has over 20% strict TDD matches. 28 developers use lenient TDD in more than 10%, but less than 20% of their intervals. 98 of the 136 developers who use lenient TDD also refactor their code according to the TDD refactoring process in Figure 4.8. For them, 48% of intervals that match against the lenient TDD are due to refactoring. Of the 136 developers, 49 have little programming experience (0–2 years), 25 have some experience (3–6 years), and the majority of 59 is very experienced (> 7 years).

In our normalized data set, the results on the use of TDD are somewhat higher, with 6% of users following strict, and 22% following lenient TDD. The distribution of testing- and refactoring is similar to the Σ values.

However, even top TDD users do not follow TDD in most sessions. For example, the user with the highest TDD usage has one session with 69% compliance to TDD. On the other hand, in the majority of the remaining sessions, the developer did not use TDD at

all (0%). We verified this to be common also for the other developers who partially used TDD. These low results on TDD are complemented by 574 projects where users *claimed* to use TDD, but in reality only 47 of the 574 *did* according to our definition.

4.4.5 RQ III.5: How Much Do Developers Test In The IDE?

In WatchDog clients, we asked developers how much time they spend on engineering tests. To compare survey answers to their actual development behavior, we consider `Reading` and `Typing` intervals, and further split the two intervals according to the type of the document the developer works on: either a production or test class. The duration of test executions does not contribute to it, as developers can typically work while tests execute. The mostly short test duration is negligible compared to the time spent on reading and typing (see Section 4.4.2). When registering new projects, developers estimated the time they spend on testing in the project. Hence, we have the possibility to verify how accurate their estimation was by comparing it to their actual testing behavior.

There are two ways to aggregate this data at different levels of granularity. The first is to explore the phenomenon on a *per-project* basis: we separately sum up the time developers are engineering (reading and writing) production classes and test classes, and divide it by the sum of the two. Then, we compare this value to the developers' estimation for the project. This way, we measure how accurate each individual prediction was. The second way is to explore the phenomenon in our whole data set, by *averaging* across project and *not* normalizing for the contributed development time (only multiplying each estimation with it).

Per-project measurement. Following Halkjelsvik et al. [214], Figure 4.13 shows the relative directional error of estimations as a histogram of the differences between the measured production percentage and its estimation *per project*. A value of 0 means that the estimation was accurate. A value of 100 denotes that the programmer expected to only work on tests, but in reality only worked on production code (-100, precisely the opposite). The picture on the correctness of estimations is diverse. In Eclipse, developers tend to overestimate their testing effort by 17%-points, see Figure 4.13a, where the median of the distribution is shifted to the right of 0, marked by the red line. While there are much fewer observations, the reverse is true for Figure 4.13c with an error of -23.4%-points. At an average estimation difference of -2.2%, IntelliJ developers seemed to be most accurate. Moreover, they have fewer extreme outliers than Eclipse (axes labels of Figure 4.13a and Figure 4.13b). However, the distribution of estimations in Figure 4.13b shows that the average mean value can be deceiving, as the graph demonstrates a broad proliferation of evening-out estimations from -40% to +50%, but no spike at 0%. There are relatively few observations for Android Studio (20) and IntelliJ (67) in comparison to Eclipse. On a per project-base, the average mean time spent testing is 28% (EC: 27%, IJ: 38%, AS: 51%, VS: 27%). However, developers estimated a distribution of 51% on production code (EC: 56%, IJ: 64%, AS: 73%), and 49% on tests, so they overestimated the time spent on testing by 21% percentage points, or 1.75 times.

Averaged measurement. When we do not normalize the data per project for our whole data set Σ, we find that all developers spend in total 89% of their time writing or reading production classes (EC: 89.3%, IJ: 98.5%, AS: 84.0% VS: 60.0%), and 11% of their time on testing (EC: 10.7%, IJ: 1.5%, AS: 16.0%, VS: 40.0%). These implausibly large differences

to the normalized testing percentage of 28% and between the IDEs remind us to consider Σ_{CN} again. Its average mean test percentage of 26.2% confirms the per-project normalized measurement we reported above (28%). We therefore use these values in the discussion.

Moreover, reading and writing are relatively uniformly spread across test and production code: while developers read production classes for 96.6% of the total time they spend in them, they read tests longer, namely 96.9% of the total time they spend in them.

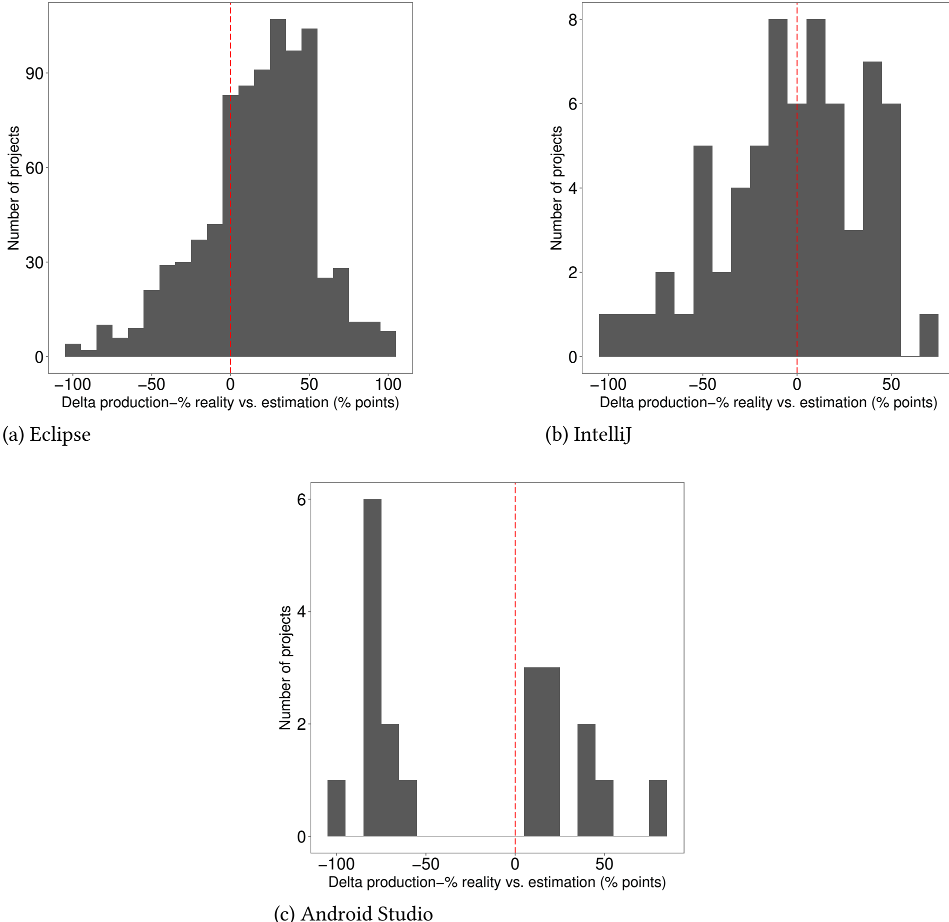


Figure 4.13: Difference between estimated and actual time spent on testing split per IDE (no data for FeedBaG++).

4.5 Discussion

In this section, we interpret the results to our research questions and put them in a broader perspective.

4.5.1 RQ III.1: Which Testing Patterns Are Common In the IDE?

In RQ III.1, we established that in over half of the projects, we did not see a single opened test, even when considering a very lenient definition that likely overestimates the number of tests. The test detection rate in the Eclipse-based client is almost twice as high as in the other clients. A possible reason might be that we concentrated our testing advertisement efforts on Eclipse. An investigation of testing practices on the popular Continuous Integration (CI) server Travis CI showed a somewhat higher test rate at 69% for Java projects [68]. Reasons might be that testing is the central phase of CI [68, 215] and that projects that have set up Travis CI might be more mature in general. This frequency is closer to the 58% we found in our normalized data set. Moreover, our IDE observation does not mean that the projects contain no tests (a repository analysis might find that there exist some), but it does indicate that testing is not a prime activity of the registered WatchDog developers. Alarmingly, only 43% of the projects that claimed to have JUnit tests in the survey actually had intervals showing tests ("truth tellers"). For the other 57%, their developer did not execute, read, or modify any test in the observation period. The varying amount of data we received from users impacts this measure, since we are more likely to detect test activity within a large amount of general activity for one user than when we have little data overall. Our data distribution suggests that normalization should give us a more realistic picture, see Figure 4.10a. Consequently, Σ_{CN} has a "truth teller" ratio of 73%. Since we likely overestimate tests, these two discoveries raise questions: Which value does testing have in practice? And, further, are (anonymous) developers' survey answers true and which measures are suitable to ensure correctness of our conclusions?

> Roughly half of projects and users do not practice testing in the IDE actively.

Only 17% of all projects comprise tests that developers can run in the IDE. The values across IDEs are relatively similar. We assume the real percentage is similar for Visual Studio, but shows lower due to the fact that tests are organized in their own project, see Section 4.1.3. For 27% of projects that have executable IDE tests developers never exercise the option to execute them. This gives a hint that testing might not be as popular as we thought [216]. Reasons might include that there are often no pre-existing tests for the developers to modify, that they are not aware of existing tests, or that testing is too time-consuming or difficult to do on a constant basis. The apparent lack of automated developer tests might be one factor for the bug-proneness of many current software systems.

Even for projects that have tests, developers did not execute them in most of the sessions. In contrast, the mean number of test runs for sessions with at least one test execution was high (15).

> Developers largely do not run tests in the IDE. However, when they do, they do it extensively.

One reason why some developers do not execute tests in the IDE is that the tests would render their machine unusable, for example during the execution of UI tests in the Eclipse

Platform UI project. The Eclipse developers push their untested changes to the Gerrit review tool [27] and rely on it to trigger the execution of the tests on the CI server. In this case, the changes only become part of the "holy repository" if the tests execute successfully. Otherwise, the developer is notified via email. Despite the tool overhead and a possibly slower reaction time, both anecdotal evidence and our low results on test executions in the IDE suggest that developers increasingly prefer such more complex setups to manually executing their tests in the IDE. IDE creators could improve the CI server support in future releases to facilitate this new workflow of developers.

Every developer is familiar with the phrase "Oops, I broke the build" [217]. The weak correlations between test churn and test executions (RQ III.1.3), and production churn and test executions (RQ III.1.4) suggest an explanation: developers simply do not assert for every change that their tests still run, because "this change cannot possibly break the tests." Even when the modifications to production or test code get larger, developers do not necessarily execute tests in the IDE more often [218]. These observations could stem from a development culture that embraces build failures and sees them as part of the normal development life cycle, especially when the changes are not yet integrated into the main development line.

The weak correlation between production and test code churn in RQ III.1.5 is, on the one hand, expected: tests often serve as documentation and specification of how production code should work, and are, therefore, less prone to change. This conclusion is in line with previous findings from repository analyses [179, 219]. If, on the other hand, a practice like TDD was widely adopted (RQ III.4), we would expect more co-evolution of tests and production code, expressed in a higher correlation. Supporting this observation, Romano et al. found that, even when following TDD, developers "write quick-and-dirty production code to pass the tests, [and] do not update their tests often" [220].

> Tests and production code do not co-evolve gracefully.

4.5.2 RQ III.2: What Characterizes The Tests Developers Run?

Another factor that could influence how often developer run tests, is how long they take to run. In RQ III.2, we found that testing in the IDE happens fast-paced. Most tests finish within five seconds, or less.

> Tests run in the IDE take a very short amount of time.

While still being fast, a notable exception to this are the tests run in Visual Studio, which took an order of magnitude longer. One reason for this could be that many C# tests might rely on additional base tests that take longer to setup. For example, the tests for FeedBaG++ require a specific base test of ReSharper, which takes 60 seconds to initialize. Another reason could be that Visual Studio facilitates debugging by running tests in the debugger automatically. Pausing on a breakpoint would be added to the tests' runtime.

We could generally not observe a relation between the test duration and their execution frequency. The reason for this could be that there is little difference between a

test that takes 0.1 seconds and one that takes 5 seconds in practice. Both give almost immediate feedback to the programmer. Hence, it seems unlikely that software engineers choose not to run tests because of their duration. Instead, our positive correlation values suggest that developers prefer tests that take slightly longer, for example because they assert more complex constructions. Thus, they might be more beneficial to developers than straight-forward, very short tests. In fact, short tests might be so limited in their coverage that developers might not find them useful enough to run them more often. This might be particularly relevant for testing mobile applications, where the typically longer running integration tests require developers to start up an Android Emulator. Our strong correlation for Android Studio suggests that developers prefer running such longer tests.

One reason for the generally short test duration is that developers typically do not execute all their tests in one test run. Instead, they practice test selection, and run only a small subset of their tests, mostly less than 1% of all available tests. This observed manual behavior differs strongly from an automated test execution as part of the build, which typically executes all tests.

Developers frequently select a specific set of tests to run in the IDE. In most cases, developers execute one test.

We can explain 99.5% of these test selections with two scenarios: developers either want to investigate a possibly failing test case in isolation (94.6% of test selections), or exclude such an irritating test case from a larger set of tests (4.9%). This finding complements and strengthens a study by Gligoric et al., who compared manual test selection in the IDE to automated test selection in a population of 14 developers [221].

4.5.3 RQ III.3: How Do Developers Manage Failing Tests?

One other possible explanation for the short time it takes tests to run in the IDE is that 65% of them fail (RQ III.3.1): once a test fails, the developer might abort the execution of the remaining tests and focus on the failing test, as discovered for RQ III.2.3.

Most test executions in the IDE fail.

This is a substantial difference to testing on Travis CI, where only 4% of Java builds fail due to failing tests [68]. For 32% of the failing test cases, we never saw a successful execution (RQ III.3.4). We built the set of tests in a project on a unique hash of their file names, which means we cannot make a connection between a failed and a successful test execution when it was renamed in-between. However, this specific scenario is rare, as observed at the commit-level by Pinto et al. [178]. Consequently, a substantial part of tests (up to 32%) are broken and not repaired immediately. As a result, developers exclude such “broken” tests from tests executions in the IDE, as observed for RQ III.2.3.

This observation motivated us to explore which test cases failures typically stem from.

Only 25% of test cases are responsible for 75% of test execution failures in the IDE.

This statement reminds us of the Pareto principle [222], the startling observation that, for many events, roughly 80% of the effects stem from 20% of the causes. The principle has been observed in Software Engineering in alike circumstances before, for example that 20% of the code contains 80% of its errors [223].

On the CI side, test executions are the main part of how fast a project builds [68]. To manage the problem of long and expensive builds, Herzig et al. built an elaborate cost model deciding which tests to skip [224]. A simulation of their model on Microsoft Windows and Office demonstrated that they would have skipped 40% of test executions. Using association rule mining based on recent historical data, such as test failure frequency, Anderson et al. demonstrated how they could reduce the duration of regression testing for another Microsoft product by also leaving out a substantial amount of tests [225]. Similarly, Figure 4.11 shows that at least in Eclipse and Android Studio, running the right 60% of test cases (and skipping 40%) results in catching all test failures. For IntelliJ and Visual Studio, the results are at a comparable ~90%. Thus, if we can select them efficiently, we can skip executing ~40% of test cases that always give a passing result in the IDE.

Both Microsoft studies have been performed on the build level, not as deep down as our findings in the "working mines of software development," the IDE. This observation on the build level trickles down to the IDE, where one would expect more changes than on the CI level. Moreover, it also shows that some tests never fail, even during the change-prone phases of development, reducing the value of such tests at least for bug-uncovering purposes.

Since test failures in the IDE are such a frequently recurring event, software engineers must have good strategies to manage and react to them.

The typical immediate reaction to a failing test is to dive into the offending test code.

All observed IDEs support this work flow by presenting the developer with the location of the test failure in the test class when double-clicking a failed execution. It is, thus, a conscious choice of the programmers to instead dive into production code (20% of reactions) that is being tested. Closing the IDE, perhaps out of frustration that the test fails, or opening the debug perspective to examine the test are very rare reactions. It is only prevalent in Figure 4.12d because Visual Studio automatically switches to this perspective when running tests. Five seconds after a test failure, ~15% of programmers have already switched focus to another application on their computer. An explanation could be that they search for a solution elsewhere, for example in a documentation PDF or on the Internet. This is useful if the test failure originates from (mis-)using a language construct, the standard library, or other well-known APIs and frameworks. Researchers try to integrate answers from internet fora such as Stack Overflow into the IDE [226], to make this possibly interrupting context switch unnecessary.

12% of test case executions show a non-deterministic result.

Flaky tests are a phenomenon that has been studied on the repository [197] and build [212, 227] level. Luo et al. classified root causes of flaky tests. They found that asynchronous waiting, concurrency, and test order dependency problems represent 77% of test flakiness causes. Including all potential factors, we have calculated a flakiness score of on average 12% of test cases per project in the IDE. A study on the flakiness of tests run on the CI server Travis CI [69] found a similar flakiness rate of 12.8% [227]. This is another instance of a finding on a build server level that seems to directly translate to the IDE of individual developers. Moreover, the test flakiness of 12% fits well to an observed reaction of (re-)executing tests 10 seconds after the initial test failure in 15% of cases for most IDEs in Figure 4.12.

Findings on the CI level on test flakiness and error responsibility seem to trickle down to the IDE of individual developers.

4.5.4 RQ III.4: Do Developers Follow TDD?

TDD is one of the most widely studied software development methodologies [198, 199, 228].[7] Even so, little research has been performed on how widespread its use is in practice. In Section 4.2.5, we developed a formal technique that can precisely measure how strictly developers follow TDD. In all our 594 projects, we found only 16 developers that employed TDD for more than 20% of their changes. Similar to RQ III.1, we notice a stark contrast between survey answers and the observed behavior of developers, even in our normalized control data set. Only in 12% of the projects in which developers claimed to do TDD, did they actually follow it (to a small degree).

According to our definition, TDD is not widely practiced. Programmers who claim to do TDD, neither follow it strictly nor for all their modifications.

The developers who partially employed TDD in our data set were more experienced in comparison to the general population. We also found a higher TDD rate in our normalized data set, likely due to the fact that Σ_{CN} has more experienced users compared to Σ and TDD followship correlates with experience.

Two recent studies support these discoveries on TDD. Borle et al. found an almost complete lack of evidence for TDD adoption in the repositories of open source GitHub projects [229]. Romano et al. found that both novice and expert programmers apply TDD in a shallow fashion even in a controlled lab experiment dedicated to TDD [230]. As a cardinal difference to our field study they found that "refactoring [...] is not performed as often as the process requires" [230], while we found developers devoting over 50% of their

[7] A Google Scholar search for "Test Driven Development" returned 15,400 hits on May, 18th, 2016, while the much older "Cleanroom Software Engineering" only returned 1,350 hits and the popular "Code Review" 17,300 hits.

TDD development intervals to the re-adoption of code. A reason might be that refactoring is inevitable in most real-world software projects, but can perhaps be avoided in a lab assignment setting.

In the following, we discuss a number of possible reasons for the apparently small adoption of TDD in practice:

1. *There is no consensus on the usefulness and value of TDD.* While there have been many controlled experiments and case studies in which TDD was found to be beneficial, there seems to be an equally high number of studies that showed no, or even adverse effects [231–233]. Moreover, some of the pro-TDD studies contradict each other on its concrete benefits: For example, Erdogmus measured that the use of TDD leads to a higher number of tests and increases productivity [234], while in a case study at IBM, TDD did not affect productivity, yet decreased the number of defects [235]. Another study at IBM and Microsoft, done in part by the same authors, found that defects decreased drastically, yet productivity declined with the introduction of TDD [236]. In light of no clear evidence for TDD, developers might simply choose not to employ it.

2. *Technical practicalities prohibit the use of TDD.* Some libraries or frameworks do not lend themselves for development in a TDD-fashion. As an example, few graphical toolkits allow development in a test-first manner for a UI.

3. *Time or cost pressure prohibits the use of TDD.* TDD is often associated with a slower initial development time, and the hope that the higher quality of code it produces offsets this cost in the longer run [237, 238]. At high pressure phases or for short-lived projects, a concise decision not to use TDD might be made.

4. *Developers might not see value in testing a certain functionality TDD-style.* We received anecdotal evidence from developers saying that they do not think the current piece of functionality they are working on mandates thorough testing, or "simply cannot be wrong."

5. *Developers skip certain phases of the required TDD process.* We received anecdotal evidence from developers saying that they "sometimes know the result of a test execution." Consequently, they might skip the mandatory test executions in Figure 4.7.

6. *The application of TDD might be unnatural.* Instead of working toward a solution, TDD puts writing its specification first. This often requires developers to specify an interface without knowing the evolution and needs of the implementation. Romano et al. accordingly report that developers found the "red" test phase of TDD, in which developers are supposed to perform the above steps, particularly challenging and demotivating [230].

7. *Developers might not know how to employ TDD.* TDD is a relatively light-weight development methodology that is taught in numerous books [5], blog posts, articles, YouTube videos, and even part of the ACM's recommendations on a curriculum for undergraduate Software Engineering programs [239]. By contrast, Janzen and Saiedian noted that one common misconception among developers was that "TDD

equals automated testing." [240] Since Beck defines TDD as "driv[ing] development with automated tests" [5], we believe practitioners have understood it correctly and that a lack of education on TDD or a wrong understanding of it is not a likely reason in most cases.

While TDD might be clear enough for all practitioners, for academic studies, we still miss a precise, formally agreed-upon definition. In fact, the lack of it might explain some of the variability in the outcomes of research on the benefits of TDD. We hope that our precise definition of TDD in terms of automata from Section 4.2.5 can help future research on a technical level.

We need to convene on a generally agreed-upon, formal definition of TDD.

In his 2014 keynote at Railsconf and subsequent blog posts [241, 242], Heinemeier Hansson sparked a debate on the usefulness and adoption of TDD, leading to a series of broadcast discussions together with Fowler and Beck on the topic "Is TDD dead?" [243]. Since our WatchDog results seemed relevant to their discussion, we approached Beck, Fowler, and Heinemeier Hansson with our paper [66] to uncover if we made any methodological mistakes, for example that our model of TDD might be erroneous. Fowler and Heinemeier Hansson replied that they were generally interested in the results of the study and identified the potential sampling bias also discussed in Section 4.6.4. Regarding the low TDD use, Fowler stated that he would not be surprised if developer testing of any kind remains uncommon.

4.5.5 RQ III.5: How Much Do Developers Test?

The question of how much time software engineers put into testing their application was first asked (and anecdotally answered) by Brooks in 1975 [181]. In contrast to our study, Brooks' numbers cover the entire development and not only software developers themselves. Nowadays, it is widely believed that "testing takes 50% of your time." While their estimation was remarkably on-par with Brooks' general estimation (average mean 50.5% production time to 49.5% test time, median 50%) in Figure 4.2, WatchDog developers tested considerably less than they thought they would at only 28% of their time, overestimating the real testing time nearly two-fold. The time developers spend testing is relatively similar across all IDEs, with the only apparent outlier of Android Studio (51%). Mobile application developers might indeed spend more time testing since the Android framework facilitates unit, integration, and UI testing ("Android Studio is designed to make testing simple. With just a few clicks, you can set up a JUnit test that runs on the local JVM or an instrumented test that runs on a device" [244]), or our developer sample from Android Studio might be too small. We need more research to better understand this phenomenon and the reasons behind it.

Developers spend a quarter of their time engineering tests in the IDE. They overestimated this number nearly twofold.

In comparison, students tested 9% of their time [65], and overestimated their testing effort threefold. Hence, real-world developers test more and have a better understanding of how much they test than students. Surprisingly, their perception is still far from reality.

The ability to accurately predict the effort and time needed for the main tasks of a software project (such as testing) is important for its coordination, planning, budgeting and, finally, successful on-time completion. In a comprehensive review of the research on human judgments of task completion durations, Halkjelsvik and Jørgensen merged the two research lines of effort prediction from engineering domains and time-duration estimation from psychology [214]. Their results showed that duration predictions frequently (more than 60% of predictions) fall outside even a 90% confidence interval given by the estimators, meaning that it is normal for predictions to be as inaccurate as observed in our study. While engineers generally seem to overestimate the duration of small tasks, they underestimate larger tasks. As testing is the smaller activity in comparison to production code for most projects (~25%:75% of work time overall), this observation fits the measured overestimation of testing effort in our study. There might be a tendency to underestimate difficult and overestimate easy tasks, particularly in software development projects [245]. As developers often dislike testing and consider it "tedious" [246, 247], this might be a contributing factor to our observed overestimation of testing. In a study on software maintenance tasks by Hatton [248], developers consistently overestimated the duration of small change requests, while they consistently underestimated larger ones. Many developers might perceive testing as the smaller task in relation to the seemingly endless complexity of coming up with a working production implementation. Consequently, Hatton's findings could help explain why our participants overestimated it. Similar to our study, Halkjelsvik and Jørgensen report that working professional engineers, while still not accurate, were better in their predictions than students [214].

A prime reason for developers' inaccurate estimations might be that predicting is an inherently difficult task, especially in fast-changing domains like software development. Another reason for the inaccuracy of predictions could be that people remember the time previous tasks took incorrectly [249]. When participants had written records of their past prediction performances, however, they became quite accurate [250]. Highly overestimating the testing effort of a software product can have adverse implications on the quality of the resulting product. Software developers should, therefore, be aware of how much they test, and how much their perception deviates from the actual effort they invest in testing in the IDE. WatchDog supplies developers with both immediate and accumulated statistics, hopefully allowing them to make more precise estimations and better planning in the future.

In conjunction with RQ III.1 and RQ III.3, our discrepancy between survey answers and real-world behavior casts doubt on whether we can trust untriaged answers from developers in surveys, especially if the respondents are unknown to the survey authors.

Objectively observed behavior in the IDE often contradicted survey answers on developers' self-estimation about testing and TDD, showcasing the importance of data triangulation.

4

4.5.6 A Note On Generality And Replicability

Long-running field studies and studies that generalize over multiple factors, such as IDEs or languages, are rare in software engineering [180, 183, 251, 252], because building and maintaining the necessary tools requires significant time efforts over prolonged periods of time. Moreover, we show that it is possible to re-cycle data that was originally not intended for this study by including the FeedBaG++ client. This chapter demonstrates that even controversial, unexpected results such as our original observations on testing patterns [65], can generalize across different state-of-the art IDEs. Our larger study – comprising ten times more data and three more IDEs – confirmed most of the observations we drew from a much shorter, less resource-intense 5-month study in only Eclipse.

> If argued correctly, even a relatively small number of observations in one environment can generalize to similar contexts in Software Engineering.

We needed to normalize only relatively few of our results, leaving most of our observations straight-forward to derive from our data. However, for some research questions, for example to counter the appearance that developers might test 20 times longer in one IDE than in another (see RQ III.5, Section 4.4.5), normalization was critical. Since filter criteria always induce a bias, this chapter also shows how an observational field study can use unfiltered and easy-to-interpret and replicate data and combine it with the smaller normalized data sample where necessary.

Our mixed-methods study also showcased the problem of reporting survey answers without further triaging. While normalizing the data improved their credibility, even with it, there was a considerable mismatch between developers' actions and their surveyed answers and beliefs. A diverse set of factors, including psychological ones, seems to play a key role in this.

4.5.7 Toward A theory of Test-Guided Development

Combining results from RQ III.1–RQ III.5, we find that most developers spend a substantial amount of their time working on codified tests, in some cases more than 50%. However, this time is shorter than expected generally and specifically by the developers themselves. Many of the tests developers work on cannot be executed in the IDE and could, therefore, not provide immediate feedback. There are relatively short development phases when programmers execute IDE-based tests heavily, followed by periods when they invoke almost no tests.

Test and production code evolution in general is only loosely coupled. This corroborates with our finding that no developer follows TDD continuously and that it, thus, seems to be a rather idealistic software development method that a small percentage of developers occasionally employs with overall little adoption in practice. We call the development practice of loosely guiding one's development with the help of tests, as the majority of developers does, relying on testing to varying degrees, *Test-Guided Development* (TGD). We argue that TGD is closer to the development reality of most programmers than TDD.

Two insights from our study, test flakiness and test failure rate, seem to be almost identical in the context of CI, showing the strong connection to individual developer testing in the IDE. However, there are also significant differences, namely that CI provides

no fast feedback loop to developers, by taking on average 20 minutes, several orders of magnitudes longer than a typical IDE test execution [68]. Test failures are much more infrequent in Java builds than in test executions in the IDE. We, therefore, argue that it plays a different, complimentary role to testing in the IDE. Due to its different and less immediate nature, CI testing cannot (fully) explain the observed low values on developer testing.

4.6 Threats to Validity

In this section, we discuss limitations and threats that can affect the validity of our study and show how we mitigated them.

4.6.1 Limitations

4

Our study has two main limitations, scope and a lack of value judgments, which we describe in the following.

Scope Definition. An endemic limitation of our study is that we can only capture what happens inside the IDE. Conversely, if developers perform work outside the IDE, we cannot record it. Examples for such behavior include pulling-in changes through an external version-control tool, such as `git` or `svn`, or modifying a file with an external editor. To reduce the likelihood and impact of such modifications, we typically limit analyses of our research questions, for example RQ III.3.5 regarding test flakiness, to one IDE session only.

Naturally, for RQ III.5, we cannot detect work on a whiteboard or thought processes of developers, which are generally hard to quantify. However, in our research questions, we are not interested in the absolute time of work processes, but in their ratio. As such, it seems reasonable to assume that work outside the IDE happens in the same ratio as in the IDE. For example, we have no indication to assume that test design requires more planning or white board time than production code.

> Our conclusions are drawn from the precisely-defined and scoped setting of codified developer testing in IDEs. To draw a holistic picture of the state of testing, we need more multi-faceted research in environments including dedicated testers.

Value Judgments. If we want to gain insight into whether more developer testing manifests in an improvement for the project, we would need to define a suitable outcome measure, for example bugs per release. One could then, for example, compare the testing effort in days across several releases, and identify whether there is a correlation. However, different projects would have different, possibly contradicting, definitions of the outcome measure: A self-managed server project in the cloud might pay no attention to bugs per release, as releases are short-lived and upgrading the software is essentially cost free. On the other hand, a server installed at a customer that cannot be reached from the outside might have this metric as its only priority. We have not defined a uniform outcome measure because (1) we could not define a sensible uniform outcome measure across all participating projects of their different priorities, (2) many developers preferred to stay anonymous, and (3) do not have or (4) would not have given us access to this highly sensible data. One can

argue that if a project reaches its desired outcome with the limited amount of testing we generally found in this study, this is better than having to spend a lot of effort on testing, it in principle wastes resources without contributing to the project's functionality. This remains a fruitful future area for deep studies on a small set of projects.

> This chapter does not contain an outcome measurement. As such, all statements are comparative to the respective groups and non-judgmental. A relative high (or low) description does not mean imply "good" or "bad."

4.6.2 Construct Validity

Construct validity concerns errors caused by the way we collect data. For capturing developers' activities we use WatchDog and FeedBaG++ (described in Section 4.1.1), which we thoroughly tested with end-to-end, integration, and developer tests. Moreover, 40 students had already used WatchDog before the start of our data collection phase [65]. Similarly, FeedBaG++ had been deployed at a company during 2015 [192] before we made it publicly available in 2016. To verify the integrity of our infrastructure and the correctness of the analysis results, we performed end-to-end tests on Linux, Windows, and MacOS with short staged development sessions, which we replicated in Eclipse, IntelliJ, and Visual Studio. We then ran our analysis pipeline and ensured the analyzed results were comparable.

When we compare data across IDEs, it is paramount that the logic that gathers and abstracts this data (to intervals) works in the same way. WatchDog's architecture with its mutually shared core guarantees this by design (see Section 4.1.1). Moreover, we had a professional software tester examine WatchDog.

To ensure the correctness of transforming FeedBaG++ events to WatchDog intervals, we implemented an extensive test suite for the transformation on the FeedBaG++ side and created a debugger that visualizes intervals similarly to the diagram shown in Figure 4.5. We used this visualization for an analysis of several manually defined usage scenarios, in which we verified that the generated intervals are accurate and that they reflect the actually recorded interactions. Moreover, we recorded artificial mini-scenarios with FeedBaG++, transferred them to WatchDog intervals and ran parts of the analysis pipeline, for example for the recognition of TDD behavior, effectively creating end-to-end tests.

4.6.3 Internal Validity

Internal validity regards threats inherent to our study.

Our study subject population shows no peculiarity (see Section 4.3.2), such as an unusually high number of users from one IP address or from a country where the software industry is weak. Combined with the fact that we use a mild form of security (HTTP access authentication), we have no reason to believe that our data has been tampered with (for example, in order to increase the chances of winning a prize).

A relatively small set of power-users contribute the majority of development sessions (Figure 4.10a). To control for the possible effects of a distorted distribution, we created a normalized data set Σ_{CN}, which showed little practical difference to our main sample.

Moreover, contrary to the idea of conducting an open field study, we run the risk of arbitrarily selecting for certain behavior by sampling. Since WatchDog and FeedBaG++ are freely available, we cannot control who installs it. Due to the way we advertise it (see Section 4.3.1), our sample might be biased toward developers who are actively interested in testing.

In the wizard in Figure 4.2 for RQ III.5, the default slider position to estimate between production and test effort was set to 50%. This could be a reason for why we received an estimation of 51%:49%. To mitigate this, WatchDog users had to move the slider before they were allowed to progress the wizard, forcing them to think about their own distribution.

The Hawthorne effect [51] poses a similar threat: participants of our study would be more prone to use, run, and edit tests than they would do in general, because they know (1) that they are being measured and (2) they can preview a limited part of their behavior. As discussed in Section 4.3.1, it was necessary to give users an incentive to install WatchDog. Without the preview functionality, we would likely not have had any users. To measure the potential impact of our immediate IDE statistics (see Figure 4.3), we tracked how often and how long developers consulted it via the `WatchDogView` interval. In total, only 192 of the 2,200 Eclipse developers opened the view in total 720 times in 422 of 39,855 possible sessions (1%), with a median open time of 2.4 minutes per user. This is similar with 58 times for 181 developers in Σ_{CN}. We believe that these measures demonstrate that developers did not constantly monitor their WatchDog recorded testing behavior, otherwise the numbers would be significantly higher. That users engage with reports about their behavior only for a short amount of time is not unique to our study: Meyer et al. found similar numbers when presenting developers with a report of their productivity [253]. Even the commercial RescueTime only had user engagement lengths of on average five seconds per day [254]. Our long observation period is another suitable countermeasure to the Hawthorne effect, as developers might change their behavior for a day, but unlikely for several months.

All internal threats point in the direction that our low results on testing are still an overestimation of the real testing practices.

4.6.4 External Validity

Threats to external validity concern the generalizability of our results. While we observed 161 years of development worktime (collected in 14,266,683 intervals originating from 2,443 developers over a period of five months), the testing practices of particular individuals, organizations, or companies are naturally going to deviate from our population phenomenon observation. Our contribution is an observation of the general state of developer testing among a large corpus of developers and projects. However, we also examined if certain sub-groups deviated significantly form our general observations. As an example of this, we identified that mainly very experienced programmers follow TDD to some extent in Section 4.4.4.

By capturing not only data from Eclipse, but also IntelliJ, Android Studio, and Visual Studio, we believe to have sufficiently excluded the threat that a certain behavior might be IDE-specific. While we have data from two programming languages (Java and C#), other programming language communities, especially non-object-oriented ones, might have different testing cultures and use other IDEs that might not facilitate testing in the same way

the Eclipse, IntelliJ, and Visual Studio IDEs do. Hence, their results might deviate from the relatively mature and test-aware Java and C# communities.

Finally, the time we measure for an activity such as testing in the IDE does not equal the effort an organization has to invest in it overall. Arguments against this are that developer testing per hour is as expensive as development (since both are done by the same set of persons), and that time is typically the critical resource in software development [214]. An in-depth investigation with management data such as real project costs is necessary to validate this in practice. To exclude the risk of a different understanding of the word testing, we specifically asked developers about JUnit testing, i.e., automated, codified developer tests (see the description in Figure 4.2).

4.7 Related Work

In this section, we first describe tools and plugins that are methodically similar to WatchDog, and then proceed with a description of related research.

4.7.1 Related Tools and Plugins

A number of tools have been developed to assess development activity at the sub-commit level. These tools include Hackystat [255], Syde [256], Spyware [257], CodingTracker [258], DFlow [259], the "Change-Oriented Programming Environment,"[8] the "Eclipse Usage Data Collector,"[9] QuantifiedDev,[10] Codealike,[11] and RescueTime.[12] However, none of these focused on time-related developer testing.

Hackystat with its Zorro extension was one of the first solutions that aimed at detecting TDD activities [260, 261], similar to the education-oriented TDD-Guide [262] and the prototype TestFirstGauge [263]. In contrast to WatchDog, Hackystat did not focus on the IDE, but offered a multitude of sensors, from bug trackers such as Bugzilla to build tool such as `ant`. One of Hackystat's challenges that we addressed with WatchDog was attracting a broader user base that allowed the recording and processing of their data.

4.7.2 Related Research

To investigate the presence or absence of tests, Kochar et al. mined 20,000 open-source projects and found that 62% contained unit tests [264]. LaToza et al. [265] surveyed 344 software engineers, testers and architects at Microsoft, with 79% of the respondents indicating that they use unit tests. Our findings indicate that only 35% of projects are concerned with testing. One factor why our figure might be smaller is that we do not simply observe the presence of some tests, but that we take into account whether they are actually being worked with.

In a study on GitHub using a repository-mining approach, Borle et al. found that a mere 3.7% of over 250,000 analyzed repositories could be classified to be using TDD [229]. This result strengthens our observed low TDD use in IDE sessions.

[8] http://cope.eecs.oregonstate.edu
[9] https://eclipse.org/epp/usagedata
[10] https://www.youtube.com/watch?v=7QKWo5SulP8
[11] https://codealike.com
[12] https://rescuetime.com

Pham et al. [266] interviewed 97 computer science students and observed that novice developer perceive testing as a secondary task. The authors conjectured that students are not motivated to test as they have not experienced its long-term benefits. Similarly, Meyer et al. found that 47 out of 379 surveyed software engineering professionals perceive tasks such as testing as unproductive [247].

Zaidman et al. [179] and Marsavina et al. [219] studied when tests are introduced and changed. They found that test and production code typically do not gracefully co-evolve. Our findings confirm this observation on a more fine-grained level. Moreover, Zaidman and Marsavina found that writing test code is phased: after a longer period of production code development, developers switch to test code. Marinescu et al. [267] observed that test coverage usually remains constant, because already existing tests execute part of the newly added code. Feldt [268] on the other hand notes that test cases "grow old": if test cases are not updated, they are less likely to identify failures. In contrast, Pinto et al. [178] found that test cases evolve over time. They highlight that tests are repaired when the production code evolves, but they also found that non-repair test modifications occurred nearly four times as frequently as test repairs. Deletions of tests are quite rare and if they happen, this is mainly due to refactoring the production code. A considerable portion of test modifications is related to the augmentation of test suites. Additionally, Athanasiou et al. investigated the quality of developer tests, noting that completeness, effectiveness, and maintainability of tests tend to vary among the observed projects [269].

The work presented in this chapter differs from the aforementioned works in that the data that we use is not obtained (1) from a software repository [178, 179, 219, 264, 268] or (2) purely by means of a survey or interview [247, 265, 266, 270]. Instead, our data is automatically gathered inside the IDE, which makes it (1) more fine-grained than commit-level activities and (2) more objective than surveys alone.

4.8 Conclusion

Our work studies how developers test in their IDE. Our goal was to uncover the underlying habits of how developers drive software development with tests. To this end, we performed a large-scale field study using low-interference observation instruments installed within the developers' working environment to extract developer activity. We complemented and contrasted these objective observations with surveys of said developers. We found that automated developer testing (at least in the IDE) is not as popular as often assumed, that developers do not test as much as they believe they do, and that TDD is not a popular development paradigm. We called the concept of loosely steering software development with the help of testing *Test-Guided Development.*

This work makes the following key contributions:

1. A low interference method and its implementation to record fine-grained activity data from within the developers' IDEs.
2. A formalized approach to detect the use of TDD.
3. A thorough statistical analysis of the activity data resulting in both qualitative and quantitative answers in developers' testing activity habits, test run frequency and time spent on testing.

4. A generalized investigation of developer testing patterns across four IDEs in two programming languages.

In general, we find a distorting gap between expectations and beliefs about how testing is done in the IDE, and the real practice. This gap manifests itself in the following implications:

Software Engineers should be aware that they tend to overestimate their testing effort and do not follow Test-Driven Development by the book. This might lead to a lower-than-expected quality in their software. Our work suggests that different tools and languages that are conceptually similar might not impact the practice as much as individuals often think, since we found few differences between data originating from them.

IDE creators could design next-generation IDEs that support developers with testing by integrating: 1) solutions from Internet fora, 2) reminders for developers to execute tests during large code changes, 3) automatic test selection, and 4) remote testing on the build server.

Researchers can acknowledge the difference between common beliefs about software testing, and our observations from studying developer testing in the real world. Specifically, there is a discrepancy between the general attention to testing and TDD in research, and their observed popularity in practice. More abstractly, developers' survey answers only partially matched their behavior in practice, and student data deviated significantly from real-world observations. This may have implications on the credibility of certain research methods in software engineering and showcases the importance of triangulation with mixed-method approaches. On a positive note, we also found that even relatively small samples from one population group might generalize well.

5

Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub

Continuous Integration (CI) has become a best practice of modern software development. Yet, at present, we have a shortfall of insight into the testing practices that are common in CI-based software development. In particular, we seek quantifiable evidence on how central testing is to the CI process, how strongly the project language influences testing, whether different integration environments are valuable and if testing on the CI can serve as a surrogate to local testing in the IDE. In an analysis of 2,640,825 Java and Ruby builds on TRAVIS CI, we find that testing is the single most important reason why builds fail. Moreover, the programming language has a strong influence on both the number of executed tests, their run time, and proneness to fail. The use of multiple integration environments leads to 10% more failures being caught at build time. However, testing on TRAVIS CI does not seem an adequate surrogate for running tests locally in the IDE. To advance research on TRAVIS CI with GITHUB, we introduce TRAVISTORRENT.

This chapter is based on

- *M. Beller, G. Gousios, and A. Zaidman. Oops, My Tests Broke The Build: An Explorative Analysis of Travis CI with GitHub, MSR'17* [68] and
- *M. Beller, G. Gousios, and A. Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration, MSR'17 (Mining Challenge Proposal)* [69].

Continuous Integration (CI) is the software engineering practice in which developers not only integrate their work into a shared mainline frequently, but also verify the quality of their contributions continuously. CI facilitates this through an automated build process that typically includes (developer) tests [185] and various static analysis tools that can be run in different integration environments [271]. Originally described by Microsoft [272] and proposed as one of the twelve Extreme Programming (XP) practices in 1997 [273], CI has become a universal industry and Open-Source Software (OSS) best practice, often used outside the context of XP [274, 275].

A full CI build comprises 1) a traditional build and compile phase, 2) a phase in which automated static analysis tools (ASATs) such as FindBugs and JSHint are run [60, 276], and 3) a testing phase, in which unit, integration, and system tests are run. If any of these three phases fails, the whole CI build is typically aborted and regarded as broken [277]. Researchers have explored the compile and ASAT phase of CI [276, 278]; yet, we still lack a quantitative empirical investigation of the testing phase to gain a holistic understanding of the CI process. This is surprising, as testing stands central in CI [271] and a better understanding is the first step to further improve both the CI process and the build tools involved.

In this chapter, we study CI-based testing in the context of Travis CI, an OSS CI as-a-service platform that tightly integrates with GitHub. While there has been research on aspects of Travis CI [279, 280], we lack an overarching explorative study to quantitatively explore the CI domain for testing from the ground up. Moreover, as accessing data from Travis CI and overlaying it with GitHub data involves difficult technicalities, researchers would profit from making this promising data source more accessible.

Our explorative research into CI is steered by five concrete propositions inspired from and raised by previous research:

P1. The use of CI is a widespread best practice. CI has become an integral quality assurance practice [281]. But just how widespread is its use in OSS projects? One study on Travis CI found an adoption rate of 45 to 90% [279]. This seems surprisingly high given it was measured in 2013, when Travis CI was still very new, and also based on only a small subset of projects.

P2. Testing is central to CI. Two studies on the impact of compilation problems and ASATs at Google found that missing dependencies are the most important reason for builds to break [276, 278]. However, these studies have not considered the testing phase of CI. To gain a complete picture of CI, we need to measure the importance and impact of the testing phase in a CI build process.

P3. Testing on the CI is language-dependent. While CI is a general purpose practice for software development projects, the programming languages used in CI have been shown to differ, e.g. in terms of programming effort [282]. As such, CI observations for one language might not generalize to other languages. A cross-language comparison might unveil which testing practices of a certain language community and culture might benefit more from CI, in terms of shorter run time or fewer broken builds.

P4. Test Integration in different environments is valuable [281, Chapter 4]. Building and integrating in different environments is time- and resource-intensive. Consequently, it should deliver additional value over a regular one-environment integration strategy. We currently lack data to support this claim.

P5. Testing on the CI is a surrogate for testing in the IDE for getting quick feedback. One of the core ideas of developer testing is to provide quick feedback to developers [283, 284]. Yet, a recent study on how 416 software developers test in their Integrated Development Environments (IDEs) [66] could not explain the "testing paradox:" developers spent a substantial 25% of their time working on tests, but rarely executed them in their IDE. We received anecdotal evidence that, instead, developers might *offload* running tests to the CI. However, it is unclear whether the CI environment is indeed a suitable replacement for running tests locally. In particular, while Fowler claims that CI provides quick feedback [271], it typically does not allow developers to execute specific tests. It also introduces other scheduling-related latencies, the magnitude of which is not known yet.

To guide our investigation of propositions P1–P5, we derived a set of research questions, presented below along with the propositions they address:

RQ IV.1 How common is Travis CI use on GitHub? (*P1*)

RQ IV.2 How central is testing to CI? (*P2, P5*)

RQ IV.2.1 How many tests are executed per build?

RQ IV.2.2 How long does it take to execute tests on the CI?

RQ IV.2.3 How much latency does CI introduce in test feedback?

RQ IV.3 How do tests influence the build result? (*P3, P4, P5*)

RQ IV.3.1 How often do tests fail?

RQ IV.3.2 How often do tests break the build?

RQ IV.3.3 Are tests a decisive part of CI?

RQ IV.3.4 Does integration in different environments lead to different test results?

Developers need to be aware of the answers to these questions to understand the *state of the art of how CI is done in the OSS community.* Especially maintainers of and newcomers to CI, whether they join an established project or plan to introduce CI, will benefit from knowing what they can expect ("How many builds are going to fail and require additional work?", "Can I use CI to execute my tests instead of locally executing them?") and how other projects are using it ("How common is CI?," "Can my project use it?"). It is thus important share our results with the community [75].

5.1 Background

In this section, we outline related CI work and build tools. We provide an overview and description of Travis CI.

5.1.1 Related Work

Introduced as one of the twelve best practices of extreme programming in 2000 [273], CI is a relatively new trend in software engineering. In their 2014 systematic review, Ståhl and Bosch provided the most recent overview over CI practices and how they differ in various settings of industrial software development [285]. Of particular interest to us is their analysis of what is considered a failure in a CI build. The most commonly observed stance is that if any test fails during the build, then the build as a whole is considered failed (e.g., [277, 286]). Ståhl and Bosch found that build failures due to test failures are sometimes accepted, however: "[I]t is fine to permit acceptance tests to break over the course of the iteration as long as the team ensures that the tests are all passing prior to the end of the iteration" [287].

A case study at Google investigated a large corpus of builds in the statically typed languages C and Java [278], uncovering several patterns of build errors. While the study is similar in nature, it focused on static compilation problems and spared out the dynamic execution part of CI, namely testing. Moreover, it is unknown whether their findings generalize to a larger set of OSS.

Vasilescu et al. examined whether a sample of 223 GitHub projects in Java, Python, and Ruby used Travis CI [279]. While more than 90% had a Travis CI configuration, only half of the projects actually used it. In follow-up research, Vasilescu et al. found that CI, such as provided through Travis CI, significantly improves their definition of project teams' productivity, without adversely affecting code quality [280].

Pinto et al. researched how test suites evolve [178]. This work is different in that we observe real test executions as they were run in-vivo on the CI server here, while Pinto et al. performed their own post-mortem, in-vitro analysis. Their approach offers a finer control over the produced log data, yet it bears the risk of skewing the original execution results, for example because a build dependency is not available anymore [178].

Pham et al. investigated the testing culture on social coding sites. In particular, they note that to nurture a project's testing culture, the testing infrastructure should be easy to set up. Their interviews furthermore lead to the observation that Travis CI "arranges for low barriers and easy communication of testing culture" [275]. By analyzing build logs, we hope to be able to see how many projects make use of this infrastructure.

With Travis CI, a public and free CI service that integrates tightly with GitHub, we have the chance to observe how CI happens in the wild on a large basis of influential OSS projects.

Similar to Travis CI, but typically setup and maintained by the project themselves, are a number of other CI servers such as CruiseControl, TeamCity, Jenkins, Hudson, and Bamboo [288].

5.1.2 Travis CI

In this section, we provide an overview over Travis CI.

Figure 5.1: Travis CI's UI for an OSS project (WatchDog, [67]).

5

Overview. Travis CI is an open-source, distributed build service that, through a tight integration with GitHub, allows projects to build and run their CI procedures without having to maintain their own infrastructure [289]. Travis CI started in 2010 as an open-source project and turned into a company in 2012. In August 2015, it supports 26 different programming languages including Java, C(++), Scala, Python, R, and Visual Basic [290]. Apart from the community edition, free to use for OSS, Travis CI also hosts a paid service that provides non-public builds for private GitHub repositories. This edition features a faster build environment [291].

User View. Figure 5.1 showcases Travis CI's main User Interface for build number 518 in the OSS project TestRoots/WatchDog [65–67]. At marker ①, we see the project name and build status of its master branch. On the right hand side ②, Travis CI shows which git commit triggered the build, its build status ("passed") along with information such as the overall build time (7 minutes, 51 seconds). The description at ③ indicates that the build was triggered by a pull request. Through link ④, we can retrieve the full history of all builds performed for this project. Under ⑤, we can see a lightly parsed and colorized dump of the log file created when executing the build. By clicking ⑥, developers can trigger a re-execution of a build.

Build Setup. Whenever a commit to any branch on a Travis CI-enabled GitHub repository is pushed, the latest commit of said branch or pull request is automatically received by Travis CI through a GitHub web hook and subsequently built. The result of this build is then displayed on GitHub. This seamless integration into projects' workflow caters for the popular pull request model [34] and is supposedly key to Travis CI's popularity among GitHub projects.

Travis CI users configure their build preferences through a top-level file in their repository. This defines the language, the default environments and possible deviations from the default build steps that Travis CI provisions for building the project. Travis CI currently only provides single-language builds, but it does support building in multiple environments, e.g., different versions of Java. For each defined build environment, Travis CI launches one job that performs the actual build work in this environment. If one of these

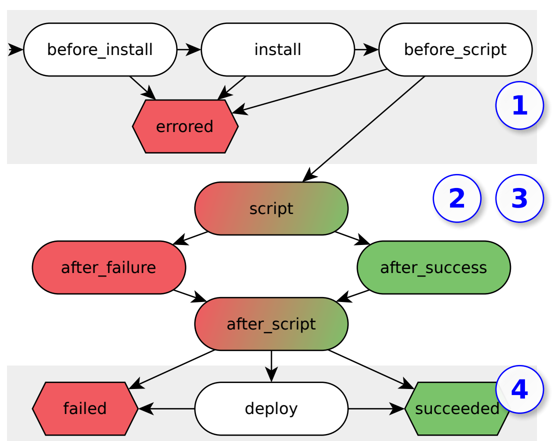


Figure 5.2: Travis CI build phases as a state machine.

jobs breaks, i.e. the build execution in one build environment exits with a non-successful status, Travis CI marks the whole build as broken. Travis CI instills a maximum job runtime of 50 minutes for OSS, after which it cancels the active job.

Build Life-cycle. Each job on Travis CI goes through the build steps depicted in the state machine in Figure 5.2: A CI build always starts with the three infrastructure provisioning phases before_ install, install and before_script. In CI phase ①, the Travis CI job is initialized, either as a legacy virtual machine like in step ⑤ in Figure 5.1 or as a new Docker container [292], the git repository is cloned, additional software packages are installed, and a system update is performed. If a problem occurs during these phases, the job is marked as *errored* and immediately aborted. The script phase actualizes the build, for example for a Java Maven project, Travis CI calls `mvn -B` to build and test the application, and for Ruby, it calls `rake` per default, CI phases ASAT and test runs, ② and ③. The build can either *succeed* or *fail*, denoted in a Unix-fashion non-zero return value from the script phase. The deploy phase ④ is optional and does not influence the build result.

Travis CI pipes the console output from the different build phases into the build log of the job, separated as so-called folds. For example, the `git.checkout` is part of the before_install phase in Figure 5.1. The output generated in the script phase contains the typical console output of build tools: build status, execution time, possible compilation problems, test executions, and failures. The format of this output depends on the actual build and test framework used.

Build Status. Travis CI features a *canceled* build status that can occur in any phase and is triggered from the outside. We call an *errored* or *failed* build more generally *broken*, opposed to a *succeeded* build.

REST API. Next to its normal user interface in Figure 5.1, Travis CI provides an unrestricted RESTful Web-API [293], using which data from all publicly built OSS repositories can be queried. The API allows us to conveniently access build-related information

to perform our deeper build analysis.

5.2 Research Setup

In this section, we give a high-level overview of our research design and describe our research methodology in detail.

5.2.1 Study Design

The main focus of this study is to evaluate how testing works in the context of CI. We performed a purely quantitative study to address our propositions, combining multiple data sources and RQs. We use the GHTorrent database [41] as a source of projects to examine and apply filtering to select the most appropriate ones. The results for RQ IV.1 lead us to the projects we would analyze in RQs IV.2 and IV.3.

These remaining research questions require a deep analysis of the project source code, process and dependency status at the job level. Moreover, as we needed to examine test tool outputs, we restricted our project search space to Ruby and Java. Both languages enjoy wide popularity among developer and have a strong testing tradition, evidenced by the plethora of available automated testing tools. Using the projects selected in the previous step as a starting point, we filtered out those that are not written in Ruby or Java and are not integrated with Travis CI. Then, we extract and analyze build information from Travis CI build logs and the GHTorrent database, combining both data sources in the newly implemented TravisTorrent.

5.2.2 Tools

In this section, we detail the tools we used to carry out our study. Our data extraction and analysis pipeline is written in Ruby and R. For replication purposes and to stimulate further research, we created TravisTorrent [69], which disseminates tools and data set on `http://travistorrent.testroots.org`.

TravisPoker. To find out which and how many projects on GitHub use Travis CI, we implemented TravisPoker. This fast and lightweight application takes a GitHub project name as input (for example, rails/rails), and finds out if and how many Travis CI builds were executed for this project.

TravisHarvester. We implemented TravisHarvester to aggregate detailed information about a project's Travis CI build history. It takes as input a GitHub project name and gathers general statistics on each build in the project's history in a CSV file. Associated with each build entry in the CSV are the SHA1 hash of the git commit, the branch and (if applicable) pull request on which the build was executed, the overall build status, the duration and starting time and the sub jobs that Travis CI executed for the different specified environments (at least one job, possibly many for each build). TravisHarvester downloads the build logs for each build for all jobs and stores them alongside the CSV file.

While both TravisPoker and TravisHarvester utilize Travis CI's Ruby client for querying the API, we could not use its job log retrieval function (`Job:log`) due to a memory leak [294] and because it does not retrieve all build logs. We circumvented these problems by also querying the Amazon AWS server that archives build logs [295].

To speed up the process of retrieving thousands of log files for each project, we parallelize our starter scripts for Travis Harvester with GNU Parallel [296].

Buildlog Analyzer. Buildlog Analyzer is a framework that supports the general-purpose analysis of Travis CI build logs and provides dedicated Java and Ruby build analyzers that parse build logs in both languages and search for output traces of common testing frameworks.

The language-agnostic Buildlog Analyzer reads-in a build log, splits it into the different build phases (folds, see Section 5.1.2), and analyzes the build status and runtime of each phase. The fold for the script phase contains the actual build and continuous testing results. The Buildlog Analyzer dispatches the automatically determined sub-Buildlog Analyzer for further examination of the build phase.

For Java, we support the three popular build tools Maven, Gradle, and Ant [297]. In Java, it is standard procedure to use JUnit as the test runner, even if the tests themselves employ other testing frameworks, such as PowerMock or Mockito. Moreover, we also support TestNG, the second most popular testing framework for Java. Running the tests of an otherwise unchanged project through Maven, Gradle and Ant leads to different, incompatible build logs, with Maven being the most verbose and Gradle the least. Hence, we need three different parsers to support the large ecosystem of popular Java build tools. As a consequence, the amount of information we can extract from a build log varies per build technology used. Some build tools give users the option to modify their console output, albeit rarely used in practice.

Example 5.1: Standard output from Maven regarding tests

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.testroots.watchdog.ClientVersionCheckerTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.04 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] All tests passed!
```

Example 5.1 shows an excerpt of one test execution from the TestRoots/WatchDog project. In the output, we can see the executed test classes (line 4), and how many tests passed, failed, errored and were skipped. We also get the test execution time (line 5). Moreover, Maven prints an overall result summary (line 9) that the Buildlog Analyzer uses to triage its prior findings. Line 11 shows the overall test execution result. Our Buildlog Analyzer gathers all this information and creates, for each invoked project, a CSV table with all build and test results for each job built. We then aggregate this information with information from the build status analyzer step by joining their output. TravisTorrent provides easy access to this data.

Example 5.2 shows the equivalent Gradle output. The silent Gradle becomes more verbose when a test fails, providing us with similar information to Example 5.1.

Example 5.2: Standard output from Gradle regarding tests

```
:test
```

By contrast, in Ruby, the test framework is responsible for the console output: it is no different to invoke RSpec through Rake than through Bundler, the two predominant Ruby build tools [297]. For Ruby, we support the prevalent Test::Unit and its offsprings, such as MiniTest. We capture behavior driven tests via RSpec and Cucumber support [298].

5.2.3 Build Linearization and Mapping to Git

If we want to answer questions such as "how much latency does CI introduce" (RQ IV.2.3), we need to make a connection between the builds performed on Travis CI and the repository which contains the commits that triggered the build. We call this build linearization and commit mapping, as we need to interpret the builds on Travis CI as a directed graph and establish a child-parent relationship based on the git commits that triggered their execution. Because there is no 1:1 relationship between builds and commits, we identified six different scenarios (a–f) arising from git's non-linear nature that make mapping a hard task. During this step, we also assessed the status of the project at the moment each build was triggered by extracting and synthesizing information from two sources: the project's git repository and its corresponding entry in the GHTorrent database.

Figure 5.3 exemplifies a typical GitHub project that uses Travis CI for its CI. In the upper part ①, we see the Travis CI builds (§1–§9), which are either passed (§1–§6, §9), canceled (§7), or broken (§8). In the lower part ②, we see the corresponding git repository hosted on GitHub with its individual commits (#A–#H). Commits #D1–#D3 live in a pull request, and not on the master branch, traditionally the main development line in git.

a) Build §1 showcases a standard situation, in which the build passed and the commit id stored with the build leads to the correct commit #A that triggered build §1. However, there are a number of more complex situations.

b) If multiple commits are transferred in one `git push` ③, only the latest of those commits is built (§2). In order to get a precise representation of the changes that lead to this build result, we have to aggregate commits #B and #C.

c) It is a central function of Travis CI to support branches or pull requests ④, such as commit #D1. When resolving builds to commits, we know from the API that §3 is a pull request build. Its associated commit points us to a virtual integration commit #V1 that is not part of the normal repository, but automatically created as a remote on GitHub ⑤. This commit #V1 has two parents: 1) the latest commit in the pull request (#D1), and 2) the current head of the branch the pull request is filed against, the latest commit on the master branch, #C. Similarly, when resolving the parent of §4, we encounter a #V2, resolve it to #D2 and the already known #C. We also know that its direct parent, #D1, is branched-off from #C. Hence, we know that any changes from build result §4 to §3 were induced by commit #D2.

d) In the case of build §6 on the same pull request ⑥, its direct predecessor is unclear: we traverse from #V3 to both 1) commit #D2 in the pull request, which is known, and to 2) #E on the master branch, which is unknown and cannot be reached from any of our previous commits #D2, #D1, or #C. This is because there was an intermediate commit #E on the master branch in-between, and pull requests are always to be integrated onto the head commit of the branch they are filed against. In such a case, one build can have multiple parents, and it is undecidable whether the changes in #D3, #E or a combination

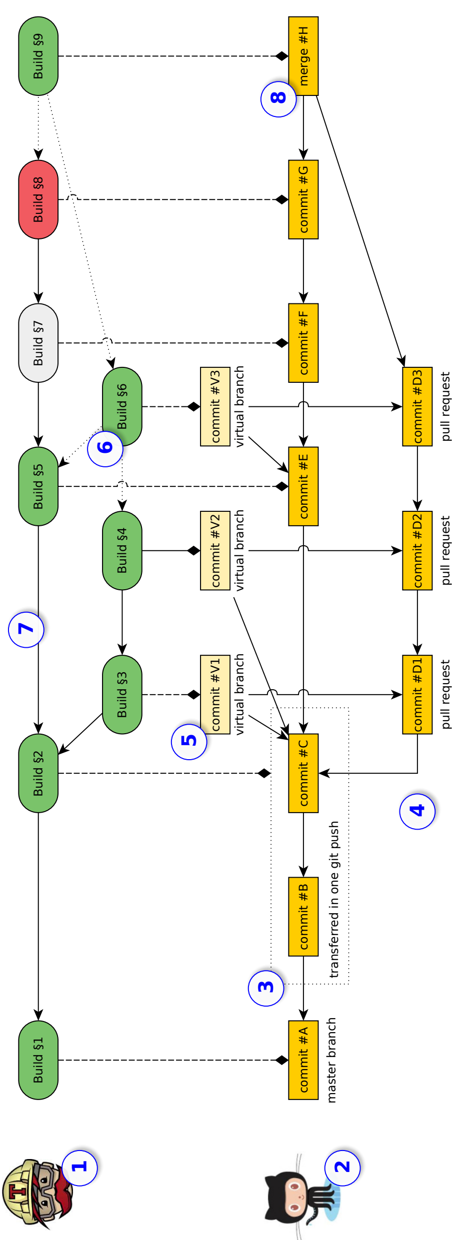


Figure 5.3: Exemplary method of how to match commits from a GitHub repository to their corresponding Travis CI builds (source: [68]).

of both lead to the build result §6.

e) Build §5 shows why a simple linearization of the build graph by its build number would fail: It would return §4 as its predecessor, when in reality, it is §2 ⑦. However, even on a single branch, there are limits to how far GIT's complex commit relationship graph can be linearized and mapped to TRAVIS CI builds. For example, if a build is canceled (§7), we do not know about its real build status – it might have passed or broken. As such, for build §8, we cannot say whether the build failure resulted from changes in commit #F or #G.

f) When merging branches or pull requests ⑧, a similar situation to c) occurs, in which one merge commit #H has two predecessors.

5.2.4 Statistical Evaluation

When applying statistical tests in the remainder of this chapter, we follow established principles [53]: we regard results as significant at a 95% confidence interval ($\alpha = 0.05$), i.e. if $p \leqslant \alpha$. All results of tests t_i in the remainder of this chapter are statistically significant at this level, i.e. $\forall i : p(t_i) \leqslant \alpha$.

For each test t_i, we perform a *Shapiro-Wilk Normality test* s_i [202]. Since all our distributions significantly deviate from a normal distribution according to it ($\forall i : p(s_i) < 0.01 \leqslant \alpha$), we use non-parametric tests: for testing whether there is a significant statistical difference between two distributions, we use the non-parametric *Wilcoxon Rank Sum test.*

Regarding effect sizes, we report Vargha-Delaney's ($\hat{A}_{12}$) [299], a non-parametric effect size for ordinal values [300]. The $\hat{A}_{12}(A, B)$ measure has an intuitive interpretation: its ratio denotes how likely distribution A outperforms B.

5.3 The TravisTorrent Data Set

In this section, we give an overview of the TRAVISTORRENT data set and ways to access it.

5.3.1 Descriptive Statistics

From the 17,313,330 active OSS repositories on GITHUB in August, 2015, our data set contains a deep analysis of the project source code, process and dependency status of 1,359 projects. To be able to do this, we restricted our project space using established filtering criteria to all non-fork, non-toy, somewhat popular (> 10 watchers on GITHUB) projects with a history of TRAVIS CI use (> 50 builds) in Ruby (936) or Java (423). Both languages are very popular on GITHUB (2nd and 3rd, respectively) [68]. Then, we extracted and analyzed build information from TRAVIS CI build logs and the GHTORRENT database for each TRAVIS CI build in its history, detailed in Section 5.3.3. Well-known projects in the TRAVISTORRENT data set include all 691,184 builds from RUBY ON RAILS, GOOGLE GUAVA and GUICE, CHEF, RSPEC, CHECKSTYLE, ASCIIDOCTOR, RUBY and TRAVIS.

5.3.2 Data-Set-as-a-Service

TRAVISTORRENT[1] provides convenient access to its archived data sets and free analytic resources: Researchers can directly access an in-browser SQL shell to run their queries on our infrastructure, and download SQL dumps or the compressed data set as a CSV file (1.8

[1] http://travistorrent.testroots.org

GB unpacked). It also provides documentation and a getting started tutorial. We share all tools we wrote to crave the data on TravisTorrent as OSS, allowing for future extensions and bug fixes by the community.

5.3.3 Data Sample

In this section, we outline all fields available in TravisTorrent and describe an abbreviated data sample.

General Data Structure. In the TravisTorrent data set, each data point (row) represents a build job executed on Travis. Every such data point synthesizes information from three different sources: The project's git repository (prefixed `git_`), data extracted from GitHub through GHTorrent (prefixed `gh_`), and data from Travis's API and an analysis of the build log (prefixed `tr_`). In total, we provide 55 data fields for each build. These are described in detail in Table 5.1.

Sample. The last column of Table 5.1 features an exemplary data point from the famous `rails/rails` project (note that there currently are 2,640,824 data points more like this in TravisTorrent). Here, we shortly highlight a few key observations.

The data sample we picked is a pretty interesting, as it is quite unusual for Rails. Not surprisingly, Rails's `project_name` is `rails/rails`. When the commit was made, 168 people had made contributions to it (it is important to realize that all metrics are calculated for the point in time in which the commit was made, so `gh_team_size` for example will grow over time). The build we are looking at (`§1543966`) comprises two commits (the latest commit built, `#c1d9c11`, and a predecessor `#87a2f021`), most likely because both commits were pushed in one go and Travis naturally builds the latest available commit. This commit is not a Pull Request (`gh_is_pr` is false), but made directly onto the stable development branch (`4-1-stable`). We could resolve a predecessor build, `§39557888`. By searching TravisTorrent for the predecessor, we could for example see whether this unusual commit directly onto the stable branch was made in order to fix an urgent problem. We can see that our BuildLogAnalyzer picked up a Ruby build with the testunit framework. 310 tests were executed successfully, until one test (SerializedAttributeTest) failed, which took 28.2 seconds (`tr_testduration`). Very unusual for Rails is that despite the failing test (`tr_tests_fail`), the overall build status was still considered passed (`tr_status`). A deeper investigation could now look into how many times this happens, and if it only occurs with specific tests, which might be ignored.

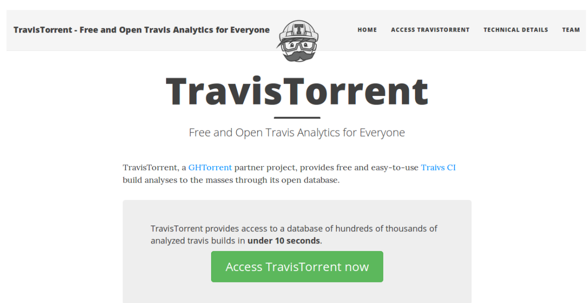


Figure 5.4: TravisTorrent (`http://travistorrent.testroots.org`) on July, 20th, 2016.

Table 5.1: Description of TravisTorrent's data fields and one sample data point from rails/rails

Column Name	Description	Unit	Example
row	Unique identifier for a build job in TravisTorrent	Integer	1543966
git_commit	SHA1 Hash of the commit which triggered this build (unique world-wide)	String	c1d9c11
git_merged_with	If this commit sits on a Pull Request (gh_is_pr true), the SHA1 of the commit that merged said pull request	String	
git_branch	Branch git_commit was committed on	String	4-1-stable
git_commits	Preceding commits that were not built (e.g., transferred in one push, ...) this build comprises	List of Strings	87a2f021
git_num_commits	The number of commits in git_commits, to ease efficient splitting	String	1
git_num_committers	Number of people who committed to this project	Integer	1
gh_project_name	Project name on GitHub (in format user/repository)	String	rails/rails
gh_is_pr	Whether this build was triggered as part of a pull request on GitHub	Boolean	false
gh_lang	Dominant repository language, according to GitHub	String	ruby
gh_first_commit_created_at	Push date of first commit in git_commits to GitHub	ISO Date (UTC+1)	2014-04-18 20:12:32
gh_team_size	Size of the team contributing to this project	Integer	168
gh_num_issue_comments	If git_commit is linked to an issue on GitHub, the number of comments on that issue	Integer	0
gh_num_commit_comments	The number of comments on git_commit on GitHub	Integer	0
gh_num_pr_comments	If gh_is_pr is true, the number of comments on this pull request	Integer	0
gh_src_churn	The churn of git_commit, i.e. how much production code changed in the commit, based on lines	Integer	4
gh_test_churn	The churn of git_commit, i.e. how much test code changed in the commit, based on lines	Integer	8
gh_files_added	Number of files added in git_commit (correlated with churn)	Integer	0
gh_files_deleted	Number of files deleted in git_commit (correlated with churn)	Integer	0
gh_files_modified	Number of files modified in git_commit (correlated with churn)	Integer	3
gh_tests_added	How many test cases were added in git_commit (e.g., for Java, this is the number of @Test annotations)	Integer	0
gh_tests_deleted	How many tests were deleted in git_commit (e.g., for Java, this is the number of @Test annotations)	Integer	0
gh_src_files	Number of production files in the repository	Integer	
gh_doc_files	Number of documentation files in the repository	Integer	
gh_other_files	Number of remaining files which are neither production code nor documentation	Integer	
gh_commits_on_files_touched	Number of commits that touched (added/deleted/modified) the files in git_commit previously	Integer	93
gh_sloc	Number of executable production source lines of code, in the entire repository	Integer	53421
gh_test_lines_per_kloc	Test density. Number of lines in test cases per 1,000 gh_sloc	Double	2191.011
gh_test_cases_per_kloc	Test density. Number of test cases per 1,000 gh_sloc	Double	188.3342
gh_asserts_cases_per_kloc	Assert density. Number of assertions per 1,000 gh_sloc	Double	535.0143
gh_by_core_team_member	Whether this commit was authored by a core team member	Boolean	true
gh_description_complexity	If gh_is_pr is true, the Pull Request's textual description complexity	Integer	
gh_pull_req_num	Pull request number on GitHub	Integer	
tr_build_id	Unique build ID on Travis	String	23298954
tr_status	Build status (pass, fail, errored, cancelled)	String	passed
tr_duration	Overall duration of the build	Integer (in seconds)	23389
tr_started_at	Start of the build process	ISO Date (UTC)	2014-04-18 19:12:32
tr_jobs	Which Travis jobs executed this build (number of integration environments)	List of Strings	[23298955, ...]
tr_build_number	Build number in the project	Integer	15459
tr_job_id	This build job's id, one of tr_jobs	String	23298981
tr_lan	Language of the build, as recognized by BuildLogAnalyzer	String	ruby
tr_setup_time	Setup time for the Travis build to start	Integer (in seconds)	0
tr_analyzer	Run BuildLogAnalyzer (ruby, java-ant, -maven, or -gradle)	String	ruby
tr_frameworks	Test frameworks that tr_analyzer recognizes and invokes (junit, rspec, cucumber, ...)	List of Strings	testunit
tr_tests_ok	If available (depends on tr_frameworks and tr_analyzer): Number of tests passed	Integer	310
tr_tests_fail	If available (depends on tr_frameworks and tr_analyzer): Number of tests failed	Integer	1
tr_tests_run	If available (depends on tr_frameworks and tr_analyzer): Number of tests were run as part of this build	Integer	311
tr_tests_skipped	If available (depends on tr_frameworks and tr_analyzer): Number of tests were skipped or ignored in the build	Integer	
tr_failed_tests	All tests that failed in this build	List of strings	Serialized-AttributeTest
tr_testduration	Time it took to run the tests	Double (in seconds)	28.2
tr_purebuildduration	Time it took to run the build (without scheduling and provisioning)	Double (in seconds)	
tr_tests_ran	Whether tests ran in this build	Boolean	true
tr_tests_failed	Whether tests failed in this build	Boolean	true
tr_num_jobs	How many jobs does this build have (length of tr_jobs)	Integer	30
tr_prev_build	Serialized link to the previous build, by giving its tr_build_id	String	39557888
tr_ci_latency	Latency induced by Travis (scheduling, build pick-up, ...)	Integer (in seconds)	1408

5.4 Results

Here, we report the results to our research questions.

5.4.1 RQ IV.1: How common is Travis CI use on GitHub?

Before investigating the testing patterns on Travis CI, we must first know 1) how many projects on GitHub use Travis CI, and 2) what characterizes them and their use of Travis CI. In August 2015, we were in a good position to measure the Travis CI adoption rate on a broad scale, as projects interested in using free CI had two years of adoption time to start to use Travis CI (see Section 5.1.2). We conjecture that, if projects have a primary interest in CI, this was enough time to hear about and set up Travis CI.

According to GHTorrent, GitHub hosted 17,313,330 active OSS repositories (including forks) in August, 2015. However, many of these 17 million projects are toy projects or duplicated (forks with no or tiny modifications). In our analysis, we are interested in state-of-the-art software systems that have a larger real-world user base. To retrieve a meaningful sample of projects from GitHub, we follow established project selection instructions [301]: we selected all projects that are not forks themselves, and received more than 50 stars.

This filtering resulted in 58,032 projects. For each project, we extracted five GitHub features from GHTorrent: main project language $\in$ {C, C++, Java, Ruby, ...}, number of watchers $\in [51; 41,663]$, number of external contributors $\in [0; 2,986]$, number of pull requests $\in [0; 27,750]$, number of issues $\in [0; 127,930]$ and active years of the project $\in [0; 45]$; using Travis Poker we collected how many Travis CI builds were executed. In total, we found that our 58,032 projects were written in 123 unique main repository languages. 16,159 projects used Travis CI for at least one build, resulting in an overall Travis CI usage rate of 27.8%. The majority of the 123 primary languages of our projects are not supported by Travis CI (see Section 5.1.2). When reducing the main project language to the 26 languages supported by Travis CI, we have 43,695 projects (75.3% of all projects). Out of these, 13,590 (31.1%) used Travis CI for at least one build.

Figure 5.5 details these findings, showing the number of projects using Travis CI aggregated per programming language. Inspired by Vasilescu et al., who found that many projects were configured for Travis CI but did not really use it, we group projects into categories with 1) no, 2) a shorter ($\leq$ 50 builds), and 3) a longer (> 50) Travis CI history. If there is a smaller number of Travis CI builds, this means that the project either recently started using Travis CI, or that Travis CI was quickly abandoned, or that the project was not active since introducing Travis CI. Due to their short build history, we have to exclude such projects from our onward analyses: it is questionable whether these projects ever managed to get CI to work properly, and if so, there is no observable history of the projects using CI. We, however, are interested in how projects work and evolve with an active use of CI.

While 31.1% is a closer approximation of the real Travis CI usage rate, Figure 5.5 hints at the fact that also projects whose main language is not supported, use Travis CI, expressed as “Other”.

In total, Travis CI executed 5,996,820 builds on all 58,032 sampled projects. Figure 5.6 gives a per-language overview of the number of builds executed per each project, based

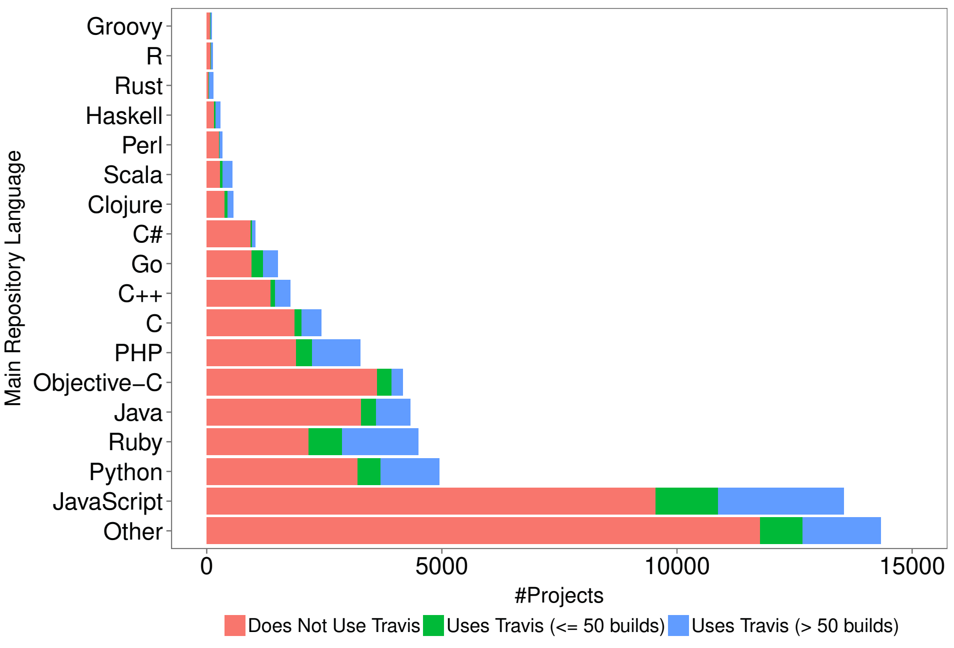


Figure 5.5: Travis CI adoption per language.

5

on all 16,159 projects that had at least one build. Next to the standard boxplot features, the ⊕-sign marks the mean number of builds.

From our 13,590 projects in a Travis CI-supported main language, we selected the ones which were best fit for an analysis of their testing patterns. We therefore ranked the languages according to their adoption of Travis CI as depicted in Figure 5.5. We also requested that they be popular, modern languages [302], have a strong testing background, one language be dynamically and the other statically typed, and the number of available projects be similar. Furthermore, there should be a fairly standard process for building and testing, widely followed within the community. This is crucial, as we must support all variants and possible log output of such frameworks. Moreover, it would be desirable if both languages show a similar build frequency in Figure 5.6. The first two languages that fulfilled these criteria were Ruby and Java. From these two languages, we sampled the 1,359 projects (Ruby: 936, Java: 423) that showed considerable Travis CI use (> 50 builds in Figure 5.5).

All further analyses are based on an in-depth investigation of 1,359 Java and Ruby projects, for which we downloaded and analyzed 2,640,825 build logs from Travis CI (1.5 TB). This is the TravisTorrent data set `travistorrent_5_3_2016`.

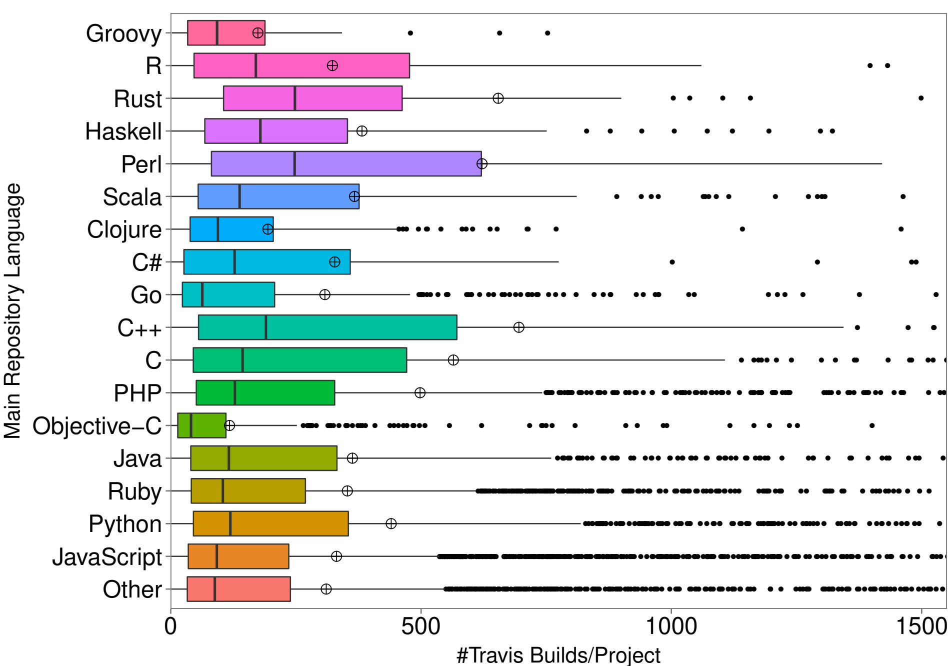


Figure 5.6: Horizontal boxplot of the #builds per project and language.

5.4.2 RQ IV.2: How central is testing to CI?

The purpose of CI is usually twofold: first, to ensure that the project build process can construct a final product out of the project's constituents and second, to execute the project's automated tests. The testing phase is not a mandatory step: in our data set, 31% of Java projects and 12.5% of the Ruby projects do not feature test runs. Overall, 81% of the projects we examined feature test runs as part of their CI process. On a per build level, 96% of the builds in our sample feature at least one test execution.

RQ IV.2.1 How many tests are executed per build? Figure 5.7 presents a histogram of the number of tests run per build on the left, and an analysis of the tests run per language on the right in the beanplot.

As expected, the histogram follows a near-power law distribution often observed in empirical studies [66]: most projects execute a smaller amount of tests, while a few run a lot of tests (mean: 1,433; median: 248; 95%: 6,779). A particular outlier in our data set was a project that consistently executed more that 700,000 tests per build. Upon manual inspection of the project (GOOGLE/GUAVA), we found that it automatically generates test cases at test runtime.

The right hand side of Figure 5.7 shows the distribution of test runs across the two programming languages in our sample; on average, Ruby builds run significantly more tests (median: 440) than Java builds (median: 15), revealed by a pairwise Wilcoxon test with a very large ($\hat{A}_{12} = 0.82$) effect size.

RQ IV.2.2: How long does it take to execute tests on the CI? Figure 5.8 depicts a

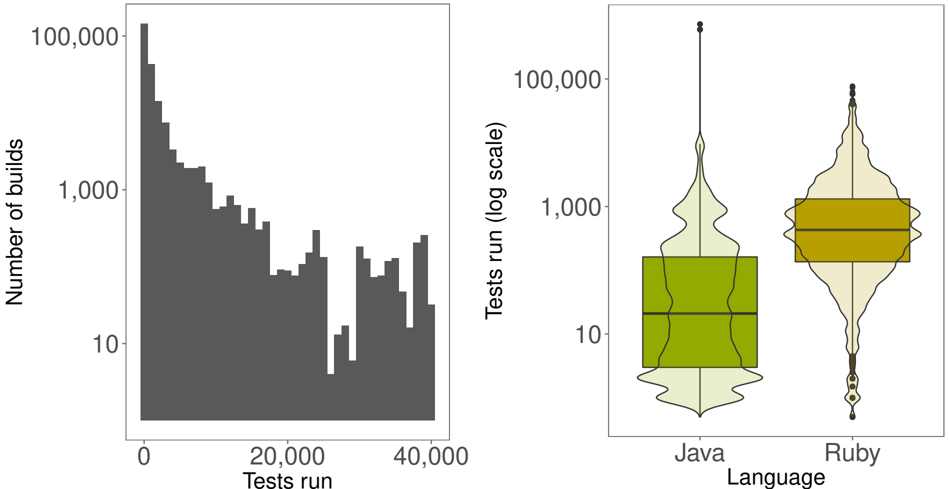


Figure 5.7: Number of tests run per build (left, log-scale) and number of tests per build per language (right, log-scale).

log-scale bean- and box-plot of the test duration, split by language. We observe that the median test duration is relatively short, at ~1 minute for Java and 10 seconds for Ruby projects. Despite the significant differences in the test duration, both languages feature a similar set of large outliers, reaching maximum test execution times of over 30 minutes.

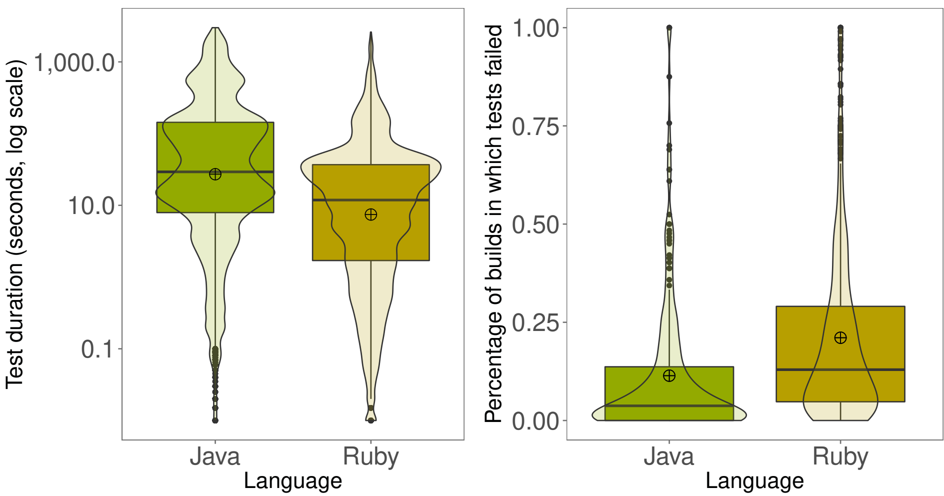
RQ IV.2.3: How much latency does CI introduce in test feedback? CI introduces a level of indirection between developers and the feedback that they receive from testing, especially when compared to testing in the IDE. In the Travis CI and GitHub setting, latency can be broken down into: the time 1) between locally committing and pushing to GitHub (commit-push latency), 2) to schedule a build after a push (job scheduling latency), 3) to provision the infrastructure (build environment setup latency), and 4) to execute the build (build execution latency). To calculate latencies, we exploit the fact that Travis CI builds are always triggered by GitHub push events. The process to connect builds to commits is as follows:

1) We identify all commits that were built on Travis CI and map commits and builds to each other (Section 5.2.3).

2) For each commit, we search for the GitHub push event that transferred it. As multiple push events might exist that contain a specific commit (e.g. a push to a repository that is cherry-picked by a fork and pushed to a different branch), we always select the earliest.

3) We list all commits in the push event and select the first one as our reference. We chose to keep the information about the first commit (and not e.g. the commit actually built) as this allows us to calculate the total latency induced by the developer (not pushing a commit creates latency to the potential feedback received by the CI) and compare it to the latency introduced by the CI.

Table 5.2 presents an overview of the latencies involved in receiving feedback from testing on the CI environment. The results reveal two interesting findings: firstly, developers tend to push their commits to the central repository shortly after they record



5

Figure 5.8: Beanplots of test duration (left) and percentage of test failures broken down per project and language (right).

Table 5.2: Descriptive statistics for CI latencies (minutes)

Latency type	5%	median	mean	80%	95%
Commit push latency	0.1	0.4	182	17.1	1,201
Job scheduling latency	0.3	1.2	33.9	8.7	85.9
Build environment setup latency	0	0	0.05	0.05	0.35
Build execution latency	0.8	8.3	19.64	30.2	77

them locally; secondly, the total time the code remains within the Travis CI environment dominates the latency time.

Developers typically push their commits quickly after their creation to the remote repository. The commit push latency distribution is very skewed; 80% of the commits only stay on the developer's local repositories for less than 17 minutes, and 50% are pushed even within one minute. The distribution skewness is a result of using distributed version control; developers have the option to commit without internet access or to delay showing their changes until perfected. Our data shows that this only happens in few cases.

On Travis CI, a build is scheduled immediately (average latency is less than a second) after commits are pushed to a repository. The time between scheduling and actual build execution depends upon resource availability for free OSS projects. The added latency is about one minute in the median case, but can reach up to nine minutes for the 80% case. While this latency is significant, it represents the price to pay for the free service offered by Travis CI; builds on commercial versions are scheduled immediately.

Moreover, before executing each build, Travis CI needs to provision a virtual machine or Docker container with the required programming language and runtime combination. This operation is usually fast: on average, across all builds, it takes 3.1 seconds (median:

0; 80%: 3; 90%: 22). However, as build job execution is mostly serial on Travis CI, the time cost to be paid is linear in the number of executed jobs or build environments. As the average project in our data set spawns 5 jobs (median: 3; 80%: 7; 90%: 10), running the build in multiple environments induces an average time overhead of 25s just for provisioning operations on the CI server.

The build process itself adds another 8.5 minutes of median latency to the test run. As there is a strict 50 minute cap on the length of build jobs, 80% of builds last 30 minutes or less.

To sum up the findings, the use of CI adds a median of 10 minutes to the time required to get feedback from testing, while the 80% case is significantly worse. The build time, which is entirely in each project's domain, dominates the feedback latency.

5.4.3 RQ IV.3: How do tests influence the build result?

With this research question, we aim to unveil how often tests fail when executed on the CI server, how often they break the build, whether they are a decisive part of CI, and if multiple build environments are useful in terms of causing different test results.

RQ IV.3.1: How often do tests fail? In RQ IV.3.1, we are interested in how often tests fail, when they are executed as part of the `script` phase of a Travis CI build.

For all 1,108 projects with test executions, Figure 5.8 shows a beanplot of the ratio of builds with at least one failed test, broken down per language. With a median of 2.9% for Java (mean: 10.3%) and a median of 12.7% (mean: 19.8%) for Ruby, the ratio of test failures among all builds is significantly higher in Ruby than in Java projects, confirmed by a Wilcoxon rank sum test with a large effect size ($\hat{A}_{12} = 0.70$).

RQ IV.3.2: How often do tests break the build? Beyond merely knowing how often tests fail, we want to research which impact this has in the bigger picture of the CI process.

Figure 5.9 shows an aggregated-per-project break-down of the build outcomes of all 1,108 projects which executed tests, separated by Java and Ruby. Next to each stacked bar, we report its participation in the overall build result. The graphs of Java and Ruby are largely comparable, with a similar build result distribution, and small differences within the groups. In total, cancels are very rare and infrastructural problems cause builds to break in around 5% of cases. Failures during the build process are responsible for most broken builds, and they are more frequent in Ruby (21.3 percentage points) than Java (14.4 % p.). In both cases, the single largest build-breaking phase is testing, with failed tests responsible for 59.0% of broken builds in Java and 52.3% in Ruby projects.

RQ IV.3.3: Are tests a decisive part of CI? Table 5.3 shows the number of builds with test failures, which have an overall failed result, and aggregates this on a per-project level. For this aggregation, a project has to consistently ignore the result of the test execution for all its history. This means that if the tests failed, this never led to a failing build. The table shows that, in general, the test execution result is decisive for the overall build result, at a per-project aggregated level of 98.3%.

Consistently ignoring the test execution result is very rare (1.7%). However, it is quite common that the failed test execution result of individual builds has no influence on the whole result (35.2%). Ignoring such individual test failures in a build is more common in Ruby (39.8%) than Java (13.6%).

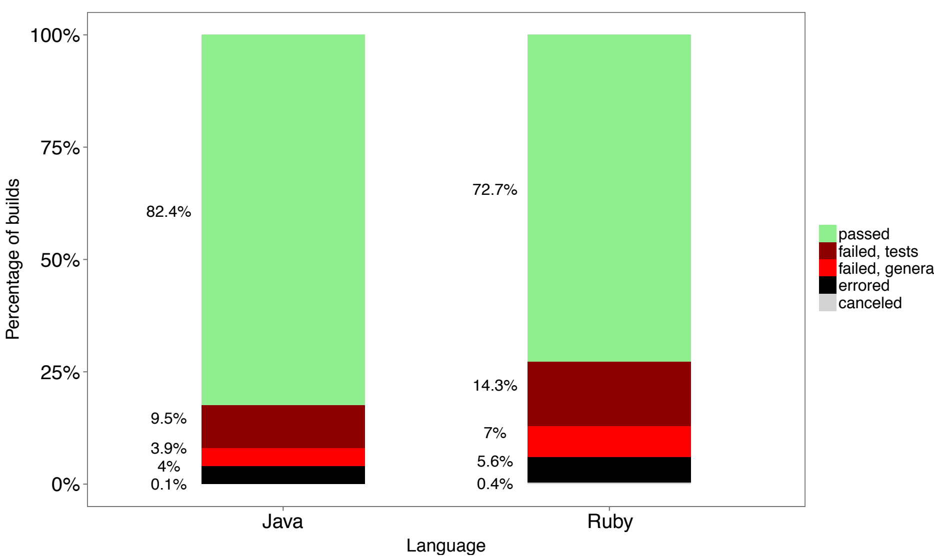


Figure 5.9: Distribution of build status per language.

Table 5.3: How often are test results ignored in a build result?

		Tests Fail → Build Fail	Tests Fail ↯ Build Pass	Total
Build level	Java	7,286 (86.4%)	1,146 (13.6%)	8,432
	Ruby	23,852 (60.2%)	15,773 (39.8%)	39,625
	Both	31,138 (64.8%)	16,919 (35.2%)	48,057
Project level	Java	197 (98.0%)	4 (2.0%)	201
	Ruby	727 (98.4%)	12 (1.6%)	739
	Both	924 (98.3%)	16 (1.7%)	940

RQ IV.3.4: Does integration in different environments lead to different test results? Each build comprises $n \geqslant 1$ job(s), which perform the same build steps in altered environments on the same GIT checkout (see Section 5.1.2). However, integration in different environments is also expensive, as the required build computation time becomes $n\times$ the individual build time. One might argue that it therefore only makes sense to do continuous integration in several environments when their execution leads to different results, capturing errors that would not have been caught with one single environment. In RQ IV.3.4, we set out to answer this question.

Table 5.4 gives an overview of how many times the execution of tests in different environments leads to a different build outcome. We observe that in total, 11.4% of builds have a different integration result, meaning that there were at least two jobs in which the test execution resulted in a different status. This effect is much more pronounced for Ruby (15.6%) than for Java (2.3%) systems. In total, over 60% of projects have at least one build in which there was a different test execution result among jobs.

Table 5.4: Results of same build in different integration environments.

Build Result		Identical	Different	Total
Jobs	Java	74,666 (97.7%)	1,725 (2.3%)	76,391
	Ruby	140,833 (84.4%)	25,979 (15.6%)	166,812
	Both	215,499 (88.6%)	27,704 (11.4%)	243,203
Projects	Java	196 (66.0%)	101 (34.0%)	297
	Ruby	240 (29.3%)	579 (70.7%)	819
	Both	436 (39.1%)	680 (60.9%)	1,116

5.5 Discussion

In this section, we discuss our propositions by combining the results of RQs IV.1–3 and then show how we mitigated threats to their validity.

5.5.1 Results

Before we started this study, we had one data point that indicated Travis CI use might be as high as 90% [279]. By contrast, regarding *P1*, we found in RQ IV.1 that around a third of all investigated projects used Travis CI in 2015. While this is significantly lower, it still shows a relatively widespread adoption. We attribute this to the facts that Travis CI provides easy-to-use default build configurations for a wide array of languages and that it is gratis.

Around 30% of GitHub OSS projects that could potentially use Travis CI for free, also make active use of it (*P1*).

Compared to about 60% of state-of-the-art GitHub projects using ASATs [60] and some 50% of projects in general doing testing [66], a number of reasons might hinder an even more-widespread use of Travis CI: Famous GitHub projects such as scala/scala [303] often run their own CI server [304] (see Section 5.1.1). This exemplifies that from the 30% adoption rate, it does not follow that 70% of projects do not use CI. For high-profile projects with a complex CI process, the migration to Travis CI would involve high initial risks and costs. One interesting line of future research therefore is to find out the overall CI adoption among top-GitHub projects. It might be that word about the benefits of CI, or Travis CI itself has not spread to every project maintainer yet.

A paradoxical finding was that a few projects written in languages that are not supported by Travis CI, still used it (Figure 5.5 "Other"). A manual investigation into a sample of such projects revealed that they also contained a smaller portion of code written in a language supported by Travis CI, for which they did enable CI. Travis CI has traditionally had deep roots in the Ruby community [305]. We found that this bias towards the Ruby community has largely vanished nowadays.

Travis CI adoption is uniform among most languages Travis CI supports (*P1*).

In RQ IV.1, we also found that the number of builds for projects varies per language, but this variation is contained. Figure 5.6 is a measure of how active projects in specific languages are and how frequently they push, thereby triggering an automated CI build and thus leveraging the full potential of CI: for each push with new commits on one branch, a new Travis CI build is triggered on the head of the branch (see Section 5.1.2). This stresses the fact that CI is a software engineering concept orthogonal to the chosen programming language used, as it applies to many languages in a similar way. Thanks to homogeneous CI use and adoption of Travis CI, researchers find a large corpus of comparable projects with similar CI patterns. This eases the interpretation of research results and decreases the need for an extensive control of external variables that would be necessary if the projects came from heterogeneous build systems. Specifically, factors such as build duration and setup process are more reliable in the homogeneous Travis CI case.

With *P2*, we were interested in the importance of testing. Overall, we found that testing stands central in CI.

> Testing happens in the majority of builds. Only ~20% of projects never included a testing phase in their CI (*P2*).

Failed tests have a higher impact, both relative and absolute and irrespective of the programming language, than compile errors, dependency resolution problems and other static checks combined. This puts the finding by Seo et al. [278] that most build errors stem from missing dependencies in perspective. Our investigation shows that issues in the compilation phase represent only a minority of the causes for broken builds (3.9% in Java, 7% in Ruby) in the bigger picture of a CI environment.

> Failing tests are the dominant reason for broken builds (*P2*).

Having established that the median number of builds in which tests fail is modest (RQ IV.3.1), but that test failures are responsible for over half of all broken builds (RQ IV.3.2), the question stands in how many instances tests fail, but the developers configured their CI in such a way that the negative test execution result does not influence the overall build status. In such cases, the test execution would not be a crucial factor to the build success. As described in Section 5.1.2, Travis CI runs tests per default and it would be a deliberate choice by developers to ignore the test result.

> Projects which consistently ignore the test result are very rare. However, ignoring individual builds is common (*P2*).

One possible reason for the difference between projects in Java and Ruby might stem from the fact that projects which do not run tests on the CI only make use of a sub-set of CI features, and therefore also have fewer builds. It might make more sense to just compile Java applications than have a CI setup for a Ruby application (that does not need to be compiled) without tests.

The typical Ruby project has ten times more tests than the typical Java project (*P2, P3*).

Given the size of our samples (423 Java and 936 Ruby projects), we believe that this difference might be attributable to the fundamental differences in the Java and Ruby programming languages. Specifically, the lack of a type system in Ruby might force developers to write more tests for what the compiler can check automatically in the case of Java [306]. We need a broader study with a larger sample of dynamic and static languages to verify whether this holds generally. With more tests, we naturally expect more test failures as a result.

Ruby projects have a four-times higher likelihood for their tests to fail in the CI environment than Java projects (*P3*).

While CI testing also happens in Java, these findings raise the question whether Java in general and JUnit test cases in particular are the best study objects for researching testing.

Having established that the large majority of projects execute tests as part of their CI, it remains to find out which indirection in the feedback cycle their execution causes (*P4*) and compare it to local test executions in the IDE (*P5*).

Multiple test environments are only useful when they also lead to different tests results in practice (*P4*). Otherwise, they just consume unnecessary resources and time. Our analysis in RQ IV.3.4 showed that test execution results vary per environment for ~10% of test executions. Some differing test results of these 10%, stem from a sheer re-execution of tests, uncovering flickering tests. One way to uncover these would be to re-execute failed builds on Travis CI and observe execution result changes. We refrained from doing so, as it would involve large costs on Travis CI.

The average project on Travis CI is tested in five integration environments (*P4*).

Our results suggest that by exposing more test failures, integration in multiple environments 1) is helpful in uncovering a substantial part of test failures and thus likely bugs that would otherwise be missed and 2) does lead to uncovering failing tests that would not be captured by running the tests locally, at the cost of an increased feedback latency. It might be more helpful for languages such as Ruby than Java.

Having established that CI adoption is relatively widespread (*P1*), that testing is integral to CI (*P2*), that it depends very much on the project language (*P3*), and that multiple integration environments are helpful in practice, it remains to discuss whether testing on the CI could replace local testing in the IDE in terms of providing quick feedback (*P5*). For this, we consider the latency induced by CI.

In RQ IV.2.3, we observed that commits only live shortly (typically, less than 20 minutes) in the developer's private repositories before developers push them upstream to the

remote mainline. Popular belief about distributed version control indicates that developers should perfect their changes locally before sharing them publicly, which one would normally assume to take longer than 20 minutes. Why do developers apparently go against this norm? Previous work on developer usage of pull requests [33, 34] might provide an indication about potential reasons. In a "fail early, fail often" approach, integrators and contributors overwhelmingly rely on their tests to assess the quality of contributed code. Instead of perfecting code in a dark corner, this has the advantage of building global awareness through communication in the pull request discussion. While collaborating developers crave for fast feedback, with 8.3 minutes build time in the median, the CI introduces measurable delays into this feedback process.

> The main factor for delayed feedback from test runs is the time required to execute the build (*P5*).

By contrast, an empirical investigation of the testing habits of 416 developers found that the median test duration ("latency") in the IDE is 0.54 seconds [66]. This is three orders of magnitude faster than running tests on the CI: testing in the IDE is fast-paced, most test executions fail (65% in comparison to 15% on the CI) and developers usually only execute one test (in contrast to all tests on the CI). Hence, the aim, the length of the feedback cycle and their observed different use in practice, suggest that testing on the CI may not be a suitable surrogate for local testing, if fast feedback is required. These findings contradict empirical evidence of projects like the "Eclipse Platform UI," which reported to increasingly offload their test executions to the CI [66]. This calls for future research: developers spend a quarter of their time working on tests [66], yet rarely execute them in the IDE. If they do, testing in the IDE is immediate and most tests fail. On the CI, most test executions pass and there is a notable delay between creating code and receiving feedback (usually, 10 minutes for pushing plus 10 minutes for building). This leaves us with the question where else developers run their tests. One such place could be building on the command line.

5.5.2 Threats to Validity

In this section, we discuss limitations and threats that affect the validity of our study, and show how we mitigated them.

Construct validity concerns errors caused by the way we collect data. For capturing build data, we relied on the custom-created tools Buildlog Analyzer, Travis Poker and Travis Harvester. To gain confidence that these tools provide us with the correct data and analysis results, we tested them with several exemplary build logs, which we also packaged into their repository and made them publicly available for replication purposes.

Threats to **internal validity** are inherent to how we performed our study. We identify two major technical threats: 1) Through the use of `git push --force`, developers can voluntarily rewrite their repository's history, leading to a wrong build linearization. By counting the number of commits that we could not resolve with the strategies described in Section 5.2.3, we received an upper bound for the severity of this threat. Since less than 10% of all commits are affected, we consider its impact small. 2) Developers can re-execute

builds (marker ⑥ in Figure 5.1), for example to re-run a build that failed due to a Travis CI infrastructural problem. In such rare cases, Travis CI overrides the previous build result. This can possibly lead to a large time difference between having pushed commits to GitHub and Travis CI starting the build, which is why we excluded the top 1% quantile of builds. As our study does not retroactively re-build projects, we are sure to observe the real build problems developers ran into.

Threats to **external validity** concern the generalizability of our results. While we examined 58,032 projects for answering RQ IV.1 and 1,359 projects for RQs IV.2–3, all projects stem from a single CI environment, namely Travis CI. We explicitly opted for Travis CI because it is frequently used along by projects hosted on GitHub, which in turn allowed us to combine two data sources and triage data such as time stamps. Nevertheless, we do not know how strongly the idiosyncrasies of both services affect the generalizability of our results on the CI-based software development model. For example, the build duration and latency measurements in RQ IV.2 depend strongly on the load and resources provided by Travis CI. When compared to other build servers, they might only be proportionally correct.

Due to the fact that we could only study the publicly visible part of Travis CI and GitHub, our study gives no indications of the practices in private projects.

Similarly, for RQ IV.2 and RQ IV.3 we only considered Java and Ruby projects to reduce the variety of build technologies and test runners. We only compared Ruby as one instance of a dynamically-typed language to Java as one instance of a statically-typed language. Some of the differences we found between them might be more attributable to the specific differences between Ruby and Java, rather than the general nature of their type system. As programming language communities have different testing cultures, an important avenue of future work is to increase the number of programming languages that are part of this investigation.

5.6 Future Work

Our work opens an array of opportunities for future work in the CI domain, both for researchers and CI tool builders.

Researchers can build on top of our results, the curated TravisTorrent data set, and open research questions. For example, we have shown that there is such significant use of Travis CI among GitHub projects that it might often be a valid shortcut to study only the streamlined Travis CI project instead of several more diverse CI sources. In particular, our investigation raises the question whether projects in dynamically-typed languages such as Ruby generally have a higher CI failure rate, and whether and which effects this has, particularly when they switch between languages. While often considered annoying and a bad smell, we do not know whether prolonged periods of broken builds or following the “break early, break often” strategy translates into worse project quality and decreased productivity. In fact, it is unclear whether breaking the build has adverse effects at all, depending on the development model. The approach proposed in Section 5.2.3 enables such studies that require previous build states. Deeper studies into integration environments could unveil how much of their uncovering 11.4% more build failures benefit is due to simply the repeated re-execution of the build (“flaky tests”).

CI Builders can use our tooling to improve their user experience. By directly indicating the nature of a build breakage, they remove the need for developers to manually inspect potentially large build logs. We strongly believe this improves developers' efficiency when dealing with failing builds.

5.7 Conclusion

In conclusion, we found that a third of popular GitHub projects make use of Travis CI, and their adoption is mostly uniform (*P1*). This finding contrasts prior research that found a far higher adoption rate of 70%. Our investigation shows that testing is an established and integral part in CI in OSS. It is the single most important reason for integration to break, more prevalent than compile errors, missing dependencies, build cancellations and provisioning problems together (*P2*). Testing is configured as a crucial factor to the success of the build, but exceptions are made on an individual level. We found that testing on the CI is highly dependent on the language, and that a dynamically typed language such as Ruby has up to ten times more tests and leads to a higher build breakage rate due to tests than a statically typed language like Java (*P3*). CI introduces a feature that local test execution cannot provide: integration in multiple environments. This is commonly used on Travis CI, and tests show different behavior when executed in multiple environments in about 10% of builds (*P4*), showcasing the value of multiple integration environments. Contrary to prior anecdotal evidence, testing on the CI does not seem a good replacement for local test executions and also does not appear to be used as such in practice (*P5*): with a latency of more than 20 minutes between writing code and receiving test feedback, the way developers use CI induces a latency that stands in contrast to the fast-paced nature of testing in the IDE and the idea that developer tests should provide quick feedback. The low test failure rates hint at the fact that developers send their contributions pre-tested to the CI server.

Apart from research on our five propositions *P1–P5*, this chapter makes the following key contributions:

- A novel method of analyzing build logs to learn about past test executions in the context of CI.
- A comparative study of CI testing patterns between a large corpus of projects written in a statically and a dynamically typed language.
- The implementation and introduction of TravisTorrent, an OSS open-access database for analyzed Travis CI build logs combined with GitHub data from GHTorrent.

6

On the Dichotomy of Debugging Behavior Among Programmers

Debugging is an inevitable activity in most software projects, often difficult and more time-consuming than expected, giving it the nickname the "dirty little secret of computer science." Surprisingly, we have little knowledge on how software engineers debug software problems in the real world, whether they use dedicated debugging tools, and how knowledgeable they are about debugging. This study aims to shed light on these aspects by following a mixed-methods research approach. We conduct an online survey capturing how 176 developers reflect on debugging. We augment this subjective survey data with objective observations on how 458 developers use the debugger included in their integrated development environments (IDEs) by instrumenting the popular ECLIPSE *and* INTELLIJ *IDEs with the purpose-built plugin* WATCHDOG *2.0. To clarify the insights and discrepancies observed in the previous steps, we followed up by conducting interviews with debugging experts and regular debugging users. Our results indicate that IDE-provided debuggers are not used as often as expected, as "printf debugging" remains a feasible choice for many programmers. Furthermore, both knowledge and use of advanced debugging features are low. These results call to strengthen hands-on debugging experience in computer science curricula and have already refined the implementation of modern IDE debuggers.*

This chapter has been published as *M. Beller, N. Spruit, D. Spinellis, and A. Zaidman. On the Dichotomy of Debugging Behavior Among Programmers, ICSE'18* [70, 193].

Debugging, the activity of identifying and fixing faults in software [307], is a tedious, but inevitable activity in almost every software development project [308]. Not only is it inevitable, but according to Kernighan and Plauger [309] and Zeller [310], so difficult that it often consumes more time than creating the bogus piece of software in the first place.

During debugging, software engineers need to relate an observed failure to its underlying defect [311]. To complete this step efficiently, they often need to acquire a deep understanding and build a mental model of the software system at hand [312]. This is where modern debuggers come in: they aid software engineers in gathering observing the system's dynamic behavior. However, they still require them to select the parts on which to focus and to perform the deductive reasoning to pinpoint the fault from the observed behaviors.

While the scientific literature is rich in terms of proposals for (automated) debugging approaches, e.g., [310, 313–316], there is a gap in knowledge of how practitioners actually debug. Debugging has thus remained *the dirty little secret of computer science* [317]. How and how much do software engineers debug at all? Do they use modern debuggers? Are they familiar with their capabilities? Which other tools and strategies do they know?

The lack of knowledge regarding developers' debugging behavior is in part due to an all too human characteristic: admitting, demonstrating, and letting others do research on how one approaches what are essentially one's own faults is a precarious situation for both a developer and a researcher. Nevertheless, by continuing to keep debugging practices secret, we miss an important opportunity for advancing software engineering theory and for delivering efficiency improvements in software development practice.

Knowledge on how developers debug can help researchers to invent more practice-relevant techniques, educators to improve their debugging curricula, and tool builders to tailor debuggers to the actual needs of developers. To open up the art of debugging, we conducted a large-scale behavioral field study on what developers think about debugging and how they debug in their IDEs. The following main questions steer our research:

RQ V.1 What do developers know about debugging and how do they reflect on it?

RQ V.2 How do developers debug in their IDEs?

RQ V.3 How do individual debugger users and experts interpret our findings from RQ V.1 and RQ V.2?

The key contributions of this chapter are:

- A triangulated, large-scale empirical study of how developers debug in reality using a mixed methods approach, supported by a replication package.[1]
- The addition of debugging features in WatchDog 2.0, an open-source, multi-platform infrastructure that allows detailed tracking of developers' debugging behavior.[2]
- Improvement suggestions for current IDE debuggers that have in part already been implemented in practice.

[1] https://archive.org/details/debugging-replication-package

[2] https://testroots.org/testroots_watchdog.html

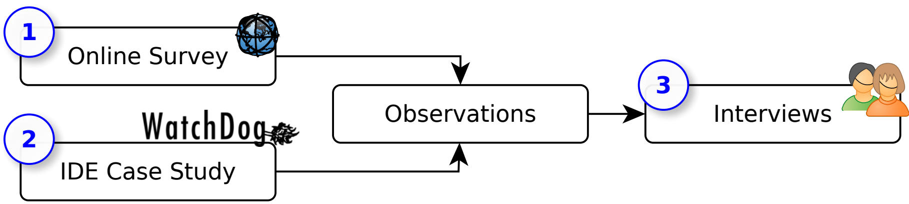


Figure 6.1: Research design overview.

Research Design

To answer these research questions, we employed a multi-faceted research approach outlined in Figure 6.1. ① We conducted an online survey to capture developers' opinions on debugging and obtained an overview of the state of the practice (see Section 6.2, Survey Results, SR). ② Simultaneously, we began using the automated WatchDog 2.0 infrastructure to track developers' fine-grained debugging activities in the IDE (see Section 6.3, WatchDog Results, WR). By instrumenting the IDE, we obtained objectively measured data, which we can compare against subjective, but richer data from the survey. We came up with a list of several, sometimes conflicting, observations that needed further explanation. ③ To help us explain the findings in depth, we conducted interviews with developers, some of whom are actively developing debugging tools (see Section 6.4, Interviews).

6.1 Related Work

Work related to our study comprises debugging tools, processes, techniques, empirical debugging evidence, and IDE instrumentation.

Debugging Tools. By "debuggers," we usually mean *symbolic debuggers*, such as the GNU Project Debugger (GDB) [318]. These debuggers allow developers to specify points in the program where the execution should halt, *breakpoints*. A typical symbolic debugger supports different *types of breakpoints*, such as *line, method, data access*, or more advanced *exception* or *class prepare breakpoints*, and options to refine the breakpoint [319]. Examples include specifying a *conditional breakpoint*, a *hit count*, a *suspension policy*, or whether the entire program or one thread should pause upon hitting a breakpoint.

Once a program halts, developers can use the symbolic debugger to permanently *watch* or ad-hoc *inspect* memory entities such as variables, work through the call stack, line-wise step through the code, or evaluate arbitrary expressions [318, 319]. Graphical debuggers such as the early *dbxtool* [320] and DDD [321] evolved from command line symbolic debuggers, such as VAX DEBUG [322], *dbx* [323], and GDB [324]. Most symbolic debugging features have since been integrated in the integrated graphical debuggers of IDEs, such as Eclipse, Visual Studio, NetBeans, and IntelliJ. This study focuses on how developers use IDE debuggers.

Debugging Process. Researchers have developed systematic process descriptions of debugging and recommendations to reduce the time programmers have to spend on finding and fixing a defect that causes a program failure. We investigate whether developers

explicitly or implicitly use debugging strategies inspired by the scientific method; for example, Zeller's *TRAFFIC* approach [310] comprises seven steps that cover every action in the debugging process, from the discovery of a problem until the correction of the defect. Three of the steps regard "the most time consuming" Find-Focus-Isolate loop, as developers often need to follow them iteratively to find the root cause of a failure. Therefore, much research has gone into techniques to, at least partially, automate this loop to reduce debugging effort [325].

In 1991, Gilmore suggested a new psychological model to understand debugging [326]. Component 1 of his model, namely that debugging is a "flexible, incomplete comprehension process [...] according to task demands, tools and skill," provides a theory-grounded description of our observations.

Automated Debugging Techniques. Arguably the most researched debugging technique is *delta debugging*, which can be used to systematically narrow down possible failure causes by comparing a successful and an erroneous program execution [327]. Other types of debugging technique include *slicing* [310], focusing on *anomalies* [310], *mining dynamic call graphs* [328], *statistical debugging* [329], *spectra-based fault localization* [313], *angelic debugging* [314], *data structure repair* [316], *relative debugging* [330], *automatic breakpoint generation* [331], *automatic program fixing using contracts* [315], and combinations thereof [332–336]. Orso presents a detailed overview of some of these automated debugging techniques [337]. However, as our study shows, automated debugging techniques have not yet reached the mainstream debugging practices and are not part of IDE debuggers. As such, we do not discuss them further.

Empirical Debugging Evidence. Only few studies exist that empirically evaluate how developers debug.

Perhaps most closely related to our study, Perscheid et al. and Siegmund et al. studied debugging practices of professional software developers [311, 338] via a survey and manual observations of each of their eight participants over "some hours during one workday" through think-aloud protocols and short interviews. Despite the fact that our studies differ significantly in population, length, and methodology, we could replicate most of their key results: the wide use of printf, a lagging adoption of advanced debugging tools and features, and developers' generally low education on debugging. We partly refined these observations, showing 1) that there is a strong dichotomy on printf use among developers, 2) which debugging features are empirically used and 3) that the complexity of operating debuggers is a main reason for these usage patterns. As in our survey, concurrency issues and external libraries seem to be the root causes of the hardest bugs. However, we also partly refuted [35]: Our developers did not run the debugger in 91% of IDE sessions and they did not spend a "huge amount of their daily work" [338] in the debugger, but less than 14%.

In a general 2006 study on how developers use Eclipse, Murphy et al. found that 90% of their 41 studied developers used the debugger [339]. This is similar to our debugger use rate in *WR1* when only considering the top 10% of users. Parnin and Rugaber observed that 13% of their 10,000 recorded sessions included debugging, compared to an IDE debugger use in 9% of the sessions in our study [340] (*WR1*).

Despite differences in study populations and methods, Layman et al. found similar

challenges and improvement wishes such as concurrency (*SQ13* in Section 6.2) and back-in-time debugging [341]. However, they do not mention some of the critical challenges found in this paper, such as debugging across languages.

Piorkowski et al. studied qualitatively how programmers forage for information [342, 343]. They found that developers spent half of their debugging time foraging for information. This complements our study as it shows what parts of the IDE are often used for finding information during debugging.

Böhme et al. studied individual steps in the debugging process, i.e., how developers localize, diagnose, and fix faults [344, 345]. Through an experiment with 12 professional software engineers they observed that fault localization is complex due to errors from an interactions of several statements. They also found that participants diagnosed bugs in a remarkably similar way. However, when fixing a fault, while the patches submitted by the participants were plausible, only 58% were correct.

IDE Instrumentation. Petrillo et al. developed the *Swarm Debug Infrastructure* (SDI), which "provides [Eclipse] tools for collecting, sharing, and retrieving debugging data" [346]. Developers can use the collective knowledge of previous debug sessions to "navigate sequences of invocation methods" and "find suitable breakpoints." SDI was evaluated in a controlled experiment involving 10 developers. Our Eclipse instrumentation for RQ V.2 is technically similar to SDI, but focuses on understanding current behavior. To increase generalizability, we also support IntelliJ and performed a longitudinal study of how dozens of developers debug in the wild.

While several WatchDog-like plugins for IDE-instrumentation exist [192, 256, 258, 261], none of them have been used to study the debugging behavior of developers, manifesting our knowledge gap of empirical debugging. Ko and Myers showed the practical usefulness of the "live IDE" wish expressed in *SQ13* in Section 6.2 with their Whyline prototype [325], which helps developers reason about assumed program behavior.

6.2 Debugging Survey

In this section, we describe our online survey.

6.2.1 Research Methods

Survey Design. To investigate developers' self-assessed knowledge on debugging for RQ V.1, we set up a survey, consisting of 13 short questions (*SQ1–SQ13*) organized in four sections; the first gathers general information about the respondents, such as programming experience and favorite IDE. The second asks if and how respondents use the IDE-provided debugging infrastructure. Developers who do not use it were asked for the reason why, while others got questions on specific debugging features, thus assessing how well the respondent knows and uses several types of breakpoints. In addition, we asked questions about other debugging features ranging from stepping through code to more advanced features like editing at run time (hot swapping). The third part, presented to all respondents, assessed the importance of codified tests in the debugging process; we gauged whether the participant uses tests for reproducing bugs, checking progress, or to verify possible bug fixes. *SQ13* was an open, non-mandatory question about participants' opinion on the statement "the best invention in debugging was printf debugging," inspired

by Brian Kernighan's quote "[t]he most effective debugging tool is still careful thought, coupled with judiciously placed print statements" [347, 348]. We included the question because research on survey design has shown that posing a concrete, controversial statement that evokes strong opinions leads to more insightful answers [349]. Before publicly releasing the survey, we sharpened it in several iterations and ran six trials with outsiders.

Card Sort. To gain an overview of the topics that concern developers, we performed an *open card sort* [350] on *SQ13*. The first two authors individually built and then mutually agreed on a set of 33 tags from a sub-sample of responses. After labeling all responses (possibly with multiple labels), the fourth author sampled 20% of the tagged responses, re-tagged them independently and compared them to the reference tag set. We then converged our tag sets into a homogeneous classification with 34 tags, agreed upon by all authors.

Dependency Analysis. To gain insights into the correlation between survey answers, we performed statistical tests. For *SQ7–12*, we had to convert each categorical answer to an ordinal scale using a linear integer transformation on its rank. This was sound because our predefined answer options have a naturally ranked order ("I don't know" = 1, "I know" = 2, ...). We then computed a pair-wise *Pearson Chi-Squared (χ^2) test of independence* [351], as we are dealing with categorical variables. If variables depended on each other ($\alpha = 0.05$), we calculated the strength of their relationship with a *Spearman rank-order correlation test* for non-parametric distributions [203]. For interpreting the results of dependency analyses ρ, we use Hopkins' guidelines [53]. They call $0 \leq |\rho| < 0.3$ no, $0.3 \leq |\rho| < 0.5$ weak, $0.5 \leq |\rho| < 0.7$ moderate and $0.7 \leq |\rho| \leq 1$ a strong correlation.

6

Subject Recruitment. To attract survey participants (*SP*), we spread the link to the survey through social media, especially Twitter, and via an in-IDE WatchDog registration dialog, advertising a raffle with three 15 Euro Amazon vouchers.

Study Subjects. We attracted 176 software developers who completed our survey. The majority of them have at least three years of experience in software development, with a third over 10 years (< 1 year: 2.8%, 1–2 years: 6.8%, 3–6: 31.8%, 7–10: 21.6%, > 10 years 36.9%). 84.1% indicated that they use Java, followed by 55.1% for JavaScript and 39.2% for Python. The languages PHP, C, C++ and C# were each selected by around 25% of participants, followed by R (16.5%), Swift (6.3%) and Objective-C (5.1%). Finally, 44 developers indicated the use of another language (24 different in total), of which Scala (11) and Ruby (8) prevail. The most used IDEs are Eclipse (31.8%), IntelliJ (30.7%), and Visual Studio (11.9%). We asked for the language to understand whether we can compare the survey results to our Java-based field study, and because certain language features define their debugging possibilities, for example the availability of a virtual machine in Java [352] or Pharo's introspection design concept, which lends itself to debugging [353, 354].

6.2.2 Results

Analysis of Survey Answers. In this section, we describe key results of our survey and RQ V.1.

SR1: Most developers use IDE debuggers in conjunction with log files and print statements. In our first question, 143 developers (81.3%) indicated that they use the IDE-provided de-

bugging infrastructure, 15 (8.5%) that they do not, and 18 (10.2%) that their selected IDE does not have a debugger. Besides using the IDE debugger, respondents indicated they examine log files (72.2%), followed closely by using print statements (71.6%). Other answers included the use of an external program (21.0%), or additional other, internal or non-generalizable techniques (30.1%). 19 developers indicated the use of a complementary method, of which adding or running tests and using web development tools built into the browser were mentioned most (both four times).

SR2: Developers not using the IDE debugging find external programs, tests, print statements, or other techniques more effective or efficient. Of the 15 developers not using the debugging infrastructure, eight think that print statements and six that techniques other than print statements are more effective or efficient. Six use an external program, while four do not know how to use a debugger.

SR3: Line breakpoints are used by the vast majority of developers. More advanced types are unknown to most. The 143 developers using an IDE debugger were asked more detailed questions on whether they know and use specific debugging features. The Likert scale plots in Figure 6.2 show that most developers are familiar with line, exception, method and field breakpoints, while temporary line breakpoints and class prepare breakpoints are known by fewer developers. The vast majority of developers also uses line breakpoints, but other breakpoint types are used by less than half of the respondents; Class prepare breakpoints are used by almost none.

SR4: Most developers answered to be familiar with breakpoint conditions, but not with hit counts and suspend policies. Figure 6.2 indicates that the majority of developers specify conditions on breakpoints. However, specifying the hit count or setting a suspend policy are both known and used less. The results in Figure 6.2 show that over 80% of the developers seem to know all major debugging features found in modern IDEs, strengthening Siegmund's findings [311]. The more advanced features, such as defining watches or a suspend policy, seem to be known and used less.

SR5: Survey answers indicate testing is an integral part of the debugging process, especially at the beginning and end. Figure 6.3 shows the use of codified tests throughout the debugging process based on all 176 responses. It indicates that tests are often used at the start and end of the debugging process, for reproducing bugs and verifying bug fixes, but slightly less during the process.

SR6: Experience has limited to no impact on the usage of the IDE-provided debugging infrastructure and tests. Examining our survey answers for dependencies allows us to understand how certain answers relate, for example whether and how strongly programmer experience correlates with the use of debugger features such as breakpoints, watches or the use of testing to guide debugging. We find that there is no correlation between the use of an IDE debugger or (unit) tests for debugging and experience in software development. There is a weak correlation between experience and specifying hit counts and a moderate correlation between experience and the usage of watches during debugging.

SR7: Developers who use tests for reproducing bugs are likely to use them for checking progress and very likely to use them for verifying bug fixes. We also find that there is a moderate correlation between the use of tests at the beginning and end of the debugging process to

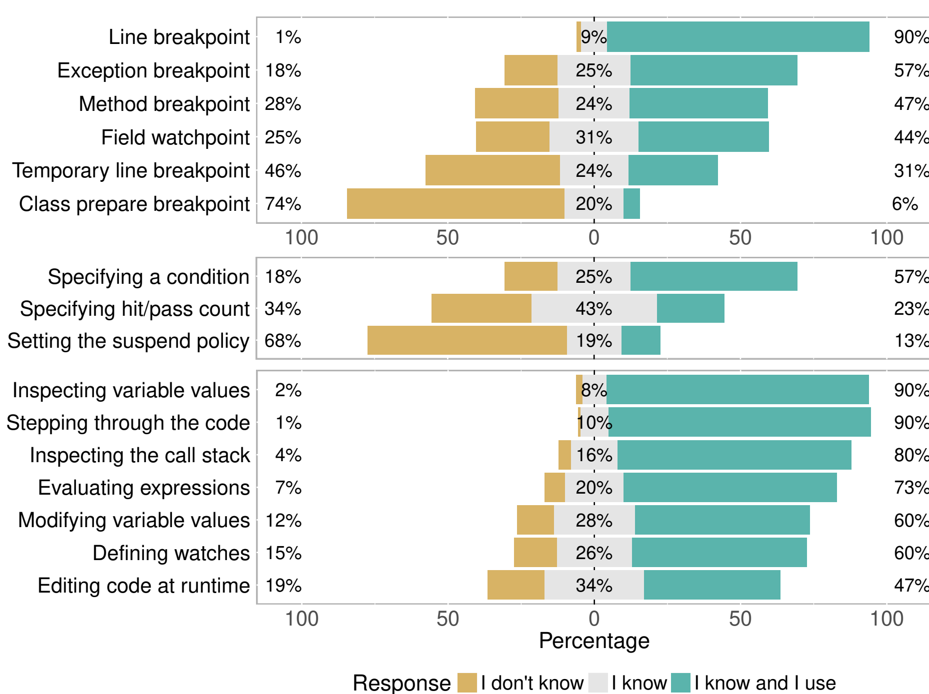


Figure 6.2: Answers in *SQ7–9* on breakpoint types, breakpoint options, and debugging features (n = 143).

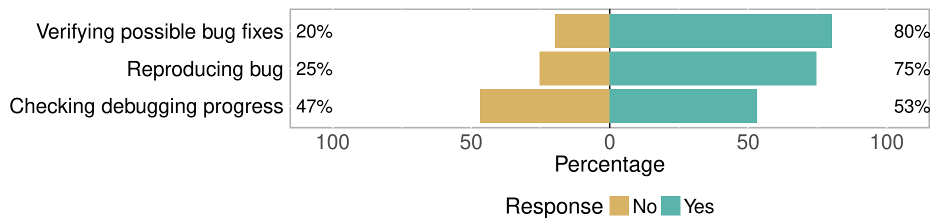


Figure 6.3: Answers in *SQ10–12* on unit tests (n = 176).

reproduce and verify bug fixes, and a weak to moderate correlation between using tests at the beginning or end and throughout the process for checking progress.

Card Sorting. In total, 108 respondents gave a response to the statement that "the best invention in debugging still was printf debugging." In the open card sorting process, we identified 34 different tags. To understand important topics and their co-occurrence, we use an intuitive graph-based representation of the tag structure. Vertices correspond to the tags and undirected, weighted edges to the strength of relation between two tags. The size of the vertices in is determined by the occurrence frequency of the tag, while the weight of the edges is determined by the relative number of co-occurrences. To ease the interpretation the graph, (1) we normalized the weights of the edges based on the occurrence frequencies of its end points, (2) we filtered out all edges with a very low normalized

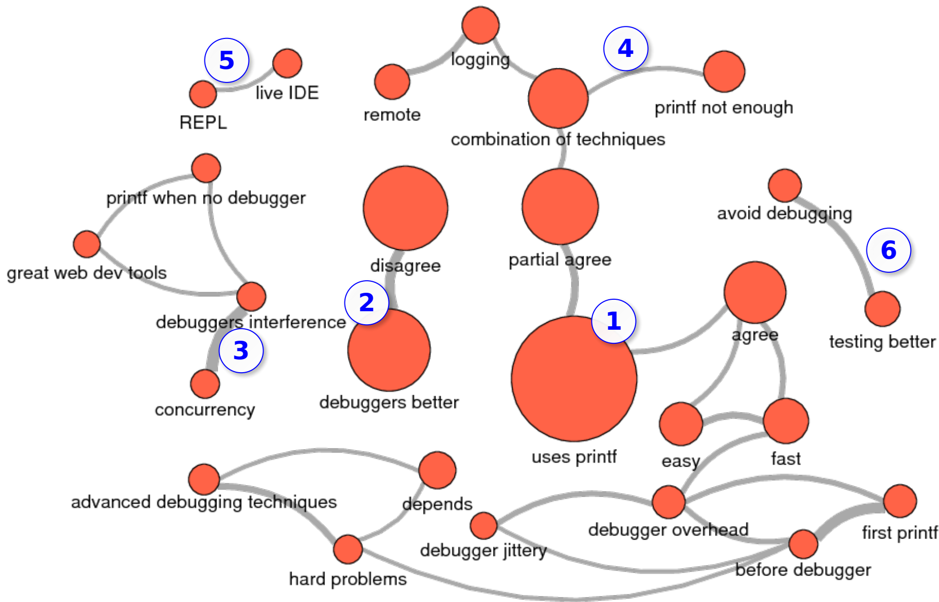


Figure 6.4: Intuitive visualization of the tag network in *SQ13*.

weight (cleaning the graph from "background noise"), and (3) we removed vertices that did not have any outgoing or incoming edge (removing unrelated concepts). The resulting graph in Figure 6.4 allows us an intuitive understanding and overview of responses and how they relate to each other, without having to read hundreds of responses [355].[3] The tags abbreviate concepts given as answers by survey respondents and are self-explanatory. The tags 'debugger jittery', 'debugger overhead' and 'debugger interference' mean that respondents think debuggers have too much impact on the thinking process, performance or program execution, respectively. 'First printf' means that developers first use printf debugging and 'before debugger' indicates that developers use some other technique(s) before using the debugger.

As the two main strongly connected subgraphs ① and ② in Figure 6.4 suggest, there was a strong dichotomy between survey respondents: Many enthusiastically agreed with our statement ("Totally agree!", *SP13, SP23*), while others rejected it, stating that "[p]eople saying that never learned how to use a debugger" (*SR54*). Developers mostly seemed to agree that IDE debuggers are methodologically superior to print statements, explaining the strong link ②. Independently, reasons for resorting back to printf are when no debugger is available or when the presence of the debugger interferes with the program execution order ③. Many of the respondents who agreed with the statement also saw drawbacks of printf debugging, like *SP10*: "Print is often most flexible but often least ef-

[3]We explicitly avoided statistical tests. Given open-ended survey answers, the meaning of such tests is unclear, or might convey a false sense of statistical precision at worst. The graph conveys our understanding having intensely worked with survey answers.

ficient." Developers indicated to use printf debugging as an ad-hoc, universal technique that is easy to do and often the first step in a possibly longer debugging strategy. However, sometimes it is not enough ④, as a combination of techniques is required. The answers also pointed to problems with IDE debuggers: they are sometimes too jittery, provide too many features and are not suited for concurrent debugging as they interfere too much. Moreover, their complicated graphical user interface (GUI) can get in the way of working (fast). Instead of printf debugging, developers seemed to prefer a live IDE with a console that has a read-eval-loop (REPL) ⑤. Summarizing this discussion, *SP75* concluded that "printf is travelling by foot, a GUI debugger is travelling [by] plane. You can go to more places by foot, but you can only go that far." Few developers also tried to avoid debugging by testing better ⑥.

6.3 IDE Field Study

In this section, we describe our field study with WatchDog 2.0.

6.3.1 Study Methods

Data Collection. To investigate the debugging habits of developers in the IDE, we extended our WatchDog infrastructure [64–67] to also track developers' debugging behavior for RQ V.2, resulting in WatchDog 2.0 for both Eclipse and IntelliJ. We had previously used WatchDog as a research vehicle to verify common expectations and beliefs about testing [64–66]. WatchDog is technically centered around the concept of *intervals* that capture the start and end of common development activities such as reading and writing code as well as running JUnit tests. We extended its interval concept to cover debugging sessions and introduce a new, orthogonal concept, singular events, that unlike intervals have no end date. Such events track when developers add, change or remove breakpoints, for example. An *IDE session* is an uninterrupted sequence of WatchDog intervals in which the developer does not close the IDE or suspend the computer. A *debugging session* is an IDE session, in which the developer used the debugger at least once.

Analysis Methods. To analyze the data collected with WatchDog 2.0, we created an open-source data processing pipeline. The pipeline, which comprises 4,000 lines of code, extracts the data from WatchDogs' MongoDB and loads it into R for further analysis. The analysis methods we used for some of these research questions require some more explanation detailed below.

For RQ V.2.1 and RQ 2.2, we assessed activity measured via WatchDog intervals. For RQ V.2.4, we assessed the intervals that occur before a debugging session is started. We chose a search range of 16 seconds before, matching the interval inactivity timeout of 16 seconds in WatchDog. This means that activity-based intervals such as reading or typing intervals are automatically closed after this period of inactivity to account for e.g. coffee breaks. A timeout length of 15 seconds is standard in IDE plugins [65, 192].

For RQ V.2.4 and RQ 2.5, we consider a file "under debugging" if we receive reading or typing intervals during a debugging interval on it, i.e. for all the files the user steps through, reads, or otherwise modifies during a debugging session.

Subject Recruitment. To attract participants to our field study, we relied on Watch-

DOG's recruitment processes [65]. Users could join or leave the field study at any time.

Study Subjects. Since the release of WATCHDOG 2.0 on 22 April 2016, we collected user data for a period over two months, until 28 June 2016. Of the 458 users, 21% come from China, 12% from India, 12% from the US, 5% from Brazil, 4% from Germany, and the remaining 46% from 65 other countries. Users could opt to share their programming experience, and 186 (41%) did: 68% had up to two years of programming experience, 16% between three to six years, and 16% seven years and more. Nine users were running MacOS X (5%), 14 Linux (6%), 162 Windows (89%), and 272 chose not to answer. Our study includes a heterogeneous mix of private, open-source, and commercial projects, with sizes ranging from green field projects to several 100,000 lines of code

The median study participation was 6 days (mean: 13 days), the maximum the full 66 days. In this period, we received 1,155,189 intervals from 458 users in 603 projects. Of these, 3,142 were debug intervals from 132 developers. In total, we recorded 18,156 hours in which the IDE was open, which amounts to 10.3 observed developer years based on the 2015 average working hours for OECD countries [356]. We also collected 54,738 debugging events from 192 users, 218 projects and 723 IDE sessions. Only 48 users ($top_{10\%}$) are responsible for 90% of the sessions, but they represent a globally diverse population with varying experience and companies working in different domains (consultancies, tool creators, financial institutions, mobile application development). In total, we recorded both at least one debug interval and one event for 108 users.

6.3.2 Results

In this section, we describe key results of our WATCHDOG 2.0 observational field study for RQ V.2.

RQ V.2.1: How prevalent and frequent is IDE debugging?

WR1: The majority of developers does not use the IDE-provided debugging infrastructure. Table 6.1 presents the number of occurrences of the different event types. Only 132 of the 458 users (28.8%) started a debugging session during the data collection period, with no significant difference between ECLIPSE (28.9%) and INTELLIJ (27.6%) users. Of these, 108 study subjects (23.6%) have used the debugger and at least one of its features (transferred both intervals and events). In $top_{10\%}$, every user had at least one debugging session (100% debugger use). No debugger use is therefore likely a result of little transferred data. However, it is not contradictory to use the debugger and not transfer any of the events listed in Table 6.1, since the debugger provides several other benefits like hot-swapping of code. In total, we observed a debugger run in 9% of all 723 IDE sessions. In the onward analyses, we only take into account data from users who used the debugger.

WR2: About 20% *of the developers are responsible for over* 80% *of the debugging intervals in our sample.* For RQ V.2.1 we are interested in knowing the frequency and length of debugging sessions. We first analyzed the number of debug intervals per user for the 132 developers that have used the debugger during the collection period. The resulting numbers range from a single debug interval to 598 debugging intervals, with an average of 23.8 and a median of 4 debug intervals per user. Next, we analyzed the duration of the 3,142 debug intervals and found values ranging from 3 milliseconds to 90.8 hours, with an

average and median duration of 13.8 minutes and 42.3 seconds, respectively. About half of the users using the IDE-provided debugging infrastructure have launched the debugger four times or less during the data collection, 21% launched their debugger more than 20 times.

RQ V.2.2: How much time is spent in IDE debugging?
WR3: Debugging consumes, on average, less than 14% of the active in-IDE development time. For RQ V.2.2, we first computed the total duration of all intervals of a particular type and based it on the total duration of 'IDE open' intervals (18,156.9 hours, 100%) in the collection period. We recorded 25.2 hours of running unit tests (0.1%), 721.5 hours of debugging intervals (4.0%), 2,568.8 hours of reading (14.1%), and 1,228.6 hours of typing (6.8%). These intervals are the main contributors of how developers spend their time in the IDE, included in the 'IDE active' intervals (28.9%). Next, we analyzed the duration and percentages on a per user basis. For the users with at least one debug interval, Table 6.2 shows the descriptive statistics of the interval duration and percentages. From the results in Table 6.2 and the fact that the total recorded active IDE time was 5250.7 hours, we conclude that debugging consumes 13.7% of the total active in-IDE development time, while reading or writing code and running tests take 48.6%, 23.4% and 0.5%.
WR4: Most debugging sessions consume less than 10 minutes. Furthermore, about half of the debugging sessions take at most 40 seconds, while about 12% of them last more than 10 minutes.

RQ V.2.3: Which IDE debugger features do developers use?
WR5: Line breakpoints are used most and by most developers, other breakpoint types are used less and by fewer developers. The results in Table 6.1 show that line breakpoints are by far the most used breakpoint type. The other, more advanced, types account for less than 7% of all breakpoints set during the collection period. Furthermore, line breakpoints are used by most developers using the debugging infrastructure, while the other types of breakpoints are used by only 7.6–20.5% of these developers.

WR6: Breakpoint options are not used by most WatchDog 2.0 users; the most frequently used option is changing their enablement. When considering how breakpoints evolve over their lifetime, the breakpoint change type frequencies in Table 6.1 (second column) indicate that almost all of these changes are related to the enablement or disablement of the breakpoints. The other change types account for only 10.9% of all breakpoint changes. Furthermore, the number of users that generated these events range from 1 (0.8%) to 12 (9.1%). Moreover, events related to specifying a hit count on the breakpoint have not been recorded during the collection period.

WR7: Setting breakpoints and stepping through code is done most, other debugging features are far less used. Table 6.1 shows that most of the recorded debugging events are related to the creation (4,544), removal (4,362) or adjustment of breakpoints, hitting them during debugging and stepping through the source code. The more advanced debugging features such as defining watches and modifying variable values have been used much less. Furthermore, the same holds for the number of users generating these events: the majority of users have added and/or removed breakpoints and stepped through the code, while only 2.3–15.2% modified variable values, evaluated expressions and/or defined watches.

RQ V.2.4: What is the relation between testing and debugging?

WR8: Most debugging sessions start after reading or changing the code, not after running tests. Regarding RQ V.2.4, we assessed the intervals that occur immediately before a debugging session starts. The resulting frequencies and their percentages of all intervals occurring before any debug interval are: 46 (0.5%) for running unit tests, 119 (1.2%) for other debug intervals, 4,991 (51.9%) for reading and 1,802 (18.7%) for typing intervals. About 70% of the debugging sessions start after reading or writing code, only 0.5% of them start after a failing or passing test run.

WR9: Developers who spend more time executing tests are likely to proportionally debug more. Next, we investigated the relation between the total duration of running unit tests and debug intervals per user. We only considered the 25 developers with at least one debug interval and one unit test execution. At $\rho = 0.58$, we find a moderate correlation between the two duration spans.

WR10: Developers who read or modify test classes longer are not likely to debug less. To complete RQ V.2.4, we studied the relation between the amount of time the user spends inside test classes (i.e., either reads or modifies tests) and the debugging time. For the 248 developers with at least one debug interval or one opened test class, we find no correlation at $\rho = -0.08$. Furthermore, we find no correlation ($\rho = 0.23$) when focusing on the 84 users with both at least one debug interval and one opened test class.

RQ V.2.5: How are file length and debugging effort related?

WR11: Smaller classes are debugged more than larger classes. Here we examined whether there is a correlation between the file size of a class (in source lines of code [357]), and the number of times the developer visits it in the source code editor in a debugging session. At $\rho = -0.75$, we find a strong negative correlation. We also investigated the relation between the file sizes and the duration of the debug intervals in which they are opened and found no apparent correlation ($\rho = 0.19$). For RQ V.2.5, we aggregated and compared the number of classes in single debug intervals to: 1. the total number of classes we observed with WatchDog for this project (also through other intervals such as reading, writing, or running tests); and 2. the number of different classes that have been debugged during any debug interval of the project.

For 1), we found that on average only 4.8% (median: 1.7%) of all project classes we observed in WatchDog intervals were ever debugged. The value ranges from 0.2% to 100%, where the 100%-cases possibly stem from small projects with only one or two classes. For 2), the results range from 0.8% to 100% with an average of 14.5% (median: 4.5%). Both results seem to indicate that debugging is focused on a relatively small set of classes in the project. In 75% of debugging sessions, at most 5% of the project's classes are debugged.

RQ V.2.6: Do developers often step over the point of interest?

WR12: Developers might step over the point of interest and have to start over again in 5% *of debugging sessions.* To answer RQ V.2.6, we first computed the total duration of all debug intervals per user. Then, we performed a Spearman rank-order correlation test using these values and the programming experience the user entered during WatchDog 2.0's registration process by applying a linear integer transformation (see Section 6.2.2). For the 58 users that have entered their experience and generated at least one debug interval,

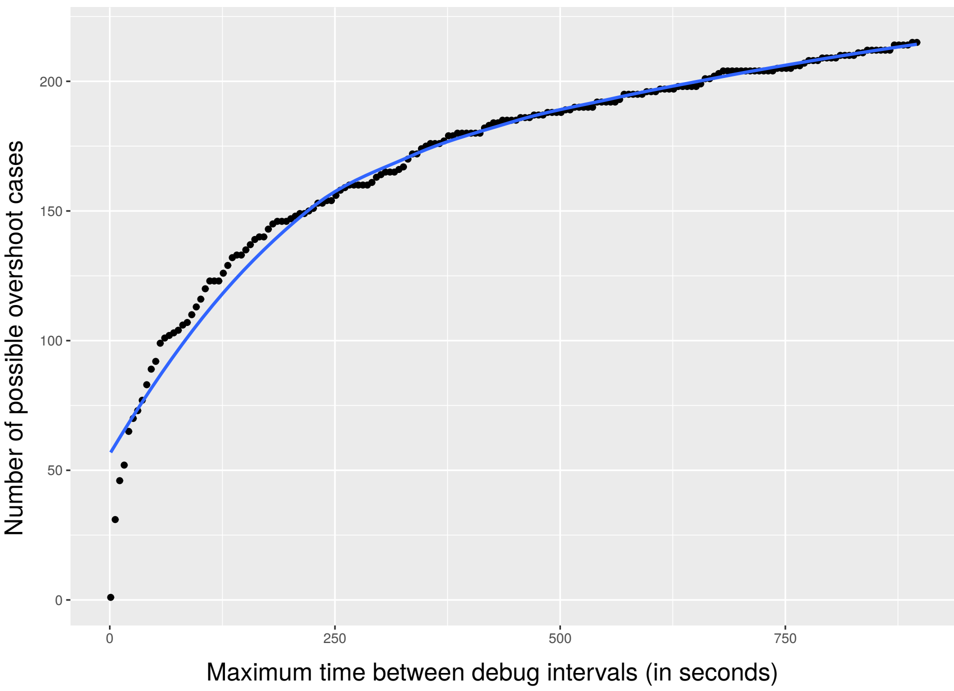


Figure 6.5: Possible cases of stepping over the point of interest per maximum time period between consecutive debug intervals.

this resulted in a weak correlation ($\rho = 0.38$), i.e. more experienced developers are more likely to spend more time in the IDE debugger.

During our research into debugging, we sometimes heard anecdotal reports of frustrated developers stepping over the point of interest while debugging. To this end, we sought objective data to support how severe the problem is by identifying possible cases of stepping over the point of interest. "Stepping over" means that the developer steps one time too far and has to start debugging all over again. Reasons for this include pressing the proceed key too fast or realizing too late that the actually interesting location was in a past step. To model this with our interval and event concept, we look for a set of debug intervals that satisfy the following conditions: 1. the last event occurring within the debug interval is a stepping event; and 2. the interval is followed by another debug interval in the same IDE session. We then created subsets of these debug intervals by imposing a maximum time t_{max} between two consecutive debug intervals. Figure 6.5 shows the possible cases of stepping over the point of interest for the subsets with $t_{max} \leq 15$ minutes.

The trend line in Figure 6.5 shows that the amount of new possible cases of stepping over the point of interest starts to decrease significantly after about four minutes. At this point, about 150 possible overshoot cases can be identified, which corresponds to 4.8% of the debugging intervals.

Table 6.1: Frequencies of breakpoint types, modifications, and WatchDog 2.0 debugging events.

Breakpoint type	Frequency	Breakpoint modification	Frequency	Event type	Frequency	Event type	Frequency
Class prepare	99	Change condition	3	Add breakpoint	4,544	(continued)	
Exception	37	Disable condition	1	Change breakpoint	247	Resume client	8,292
Field	78	Enable condition	19	Remove breakpoint	4,362	Suspend by breakpoint	13,276
Line	4,229	Disable	180	Define watch	343	Suspend by client	16
Method	77	Enable	40	Evaluate expression	101	Step into	3,480
Undefined	24	Change suspend policy	4	Inspect variable	179	Step over	19,543
				Modify variable value	4	Step out	351
	Σ 4,544		**Σ 247**	(continuing ...)			**Σ 54,738**

Table 6.2: Descriptive usage statistics for key interval types (relative to total observed time).

Variable	Unit	Min	25%	Median	Mean	75%	Max	Log-Histogram
Debugging	Hours (%)	0.00 (0.0%)	0.03 (0.1%)	0.30 (0.5%)	5.47 (2.5%)	1.42 (2.4%)	333.70 (30.8%)	
Reading	Hours (%)	0.00 (0.0%)	0.14 (1.7%)	0.60 (3.2%)	5.70 (4.9%)	2.07 (5.7%)	591.10 (52.7%)	
Typing	Hours (%)	0.00 (0.0%)	0.21 (1.5%)	1.01 (3.6%)	2.95 (4.8%)	2.78 (6.9%)	63.87 (28.3%)	
Running JUnit tests	Hours (%)	0.00 (0.0%)	0.00 (0.0%)	0.01 (0.0%)	0.68 (0.2%)	0.56 (0.2%)	9.19 (2.1%)	

6.4 Interviews

In this section, we describe how we conducted developer interviews for RQ V.3 and merge and discuss results from RQ V.1 and RQ V.2.

6.4.1 Study Methods

Interview Design & Method. To validate and obtain a deeper understanding of our findings from RQ V.1 and RQ V.2 and to mitigate apparent controversies, we ran the combined observations from survey, objective IDE measurements, and anecdotal interview insights across two sets of debugging experts. A question sheet helped us steer the semi-structured interviews, which we conducted remotely via Skype and took from 36 minutes to 67 minutes. In one case (*E3*), we performed the interview asynchronously via email. Subsequently, we transcribed the interviews and extracted insightful quotes.

Study Subjects. Table 6.3 gives an overview of our nine interviewees. We sampled the set of "regular developers" from our survey population to gain insights into what hinders the use of debuggers, why printf debugging is still widely used, and whether they regularly step over the line of interest. We chose the experts based on their industrial and academic position in the debugging field.

6.4.2 Results

This section juxtaposes survey (RQ V.1) and IDE study (RQ V.2) results and discusses them with the qualitative insights from RQ V.3.

Use of the IDE Debugger. In *WR1*, we found that two thirds of the WatchDog 2.0 users were not using the IDE-provided debugger in our observation period, an obvious contradiction to *SR1,* in which 80% of respondents claimed to use it. Moreover, no single user spent more than 30% of his development time debugging. There might be several reasons for the discrepancy: 1) The study populations are different, and the survey respondents were likely self-selecting on their interest in debugging, resulting in a higher than real use of the debugger. 2) As often observed in user studies, most relevant data stems from a relatively small percentage of users. 3) WatchDog users were free to start and stop using the plugin at any time in the observation period. Hence, for some users the actual observation period might be much shorter, perhaps coinciding with not having to debug a problem. 4)

Table 6.3: Interviewed developers and debugging experts

ID	Occupation	Dev. Experience	Country	Area
I1	Freelancer	> 20 years	Germany	Rich Client Platforms
I2	Developer	≥ 15 years	India	E-commerce
I3	Developer	11 years	USA	Real-Time Systems
I4	Developer	10 years	UK	Data Scraping
E1	3 Eclipse Debugging Project Leaders		Switzerland, India	Eclipse Development
E2	Professor	> 20 years	Greece	Software Engineering
E3	Debugger Developer	18 years	Russia	IDE Development

Almost equally many developers conceded to use printf statements for debugging in *SR2*. We have anecdotal evidence from RQ V.3 that they might use them even more: When we asked *I3* about printf debugging, he was very negative about it. Later in the interview, he still conceded to use printf "very rarely." We believe a similar observation might hold for many WatchDog users. As we cannot capture printf debugging or debugging outside the IDE with WatchDog, our finding does not mean two thirds of developers did not debug. 5) The phenomenon of a discrepancy between survey answers and observed behavior is not new. Beller et al. observed a similar phenomenon with developers claiming to spend more time on testing than they really were [66]. As a consequence, we emphasize their finding that survey answers always be cross-validated by other methods.

Printf Debugging. From RQ V.1 and RQ V.2, it seemed that developers were well-informed about printf debugging and that it is a conscious choice if they employ it, often the beginning of a longer debugging process. Interviewees praised printf as a universal tool that one can always resort back to, helpful when learning a new language ecosystem, in which one is not yet familiar with the tools of the trade. About left-over print statements that escape to production, I2 was "not worried at all, because we have a rigorous code review process." While frequently used, developers are also aware of its shortcomings, saying that "you are half-way toward either telemetry or toward tracing" and "that it is insufficient for concurrent programs, primarily because the [output] interleave[s] in strange ways" (*I3*).

> There is a strong dichotomy between developers on whether to use printf debugging.

6

Use of Debugging Features. *SR3* and *SR4* indicated that most developers use line breakpoints, but do not use more advanced breakpoint types like class prepare breakpoints. While many developers knew and used conditional breakpoints, they were widely ignorant of hit counts and the debugger's other more advanced functions. *WR5* to *WR7* support this result, finding that conditional breakpoints are indeed the second most used feature in the IDE debugger. A similar result is visible in other debugging features such as stepping through code. In both cases we found that these features get used less as they become more advanced. However, the observed numbers on the use of these features are much lower than the claimed usage visualized in Figure 6.2. For example, while 60% of the survey respondents indicated to define watches during debugging, only 15.2% of the WatchDog 2.0 users who use the debugger have defined a watched expression. Through our interviews with the debugging experts, we identify three possible causes for this.

1) More advanced debugging features are seldom required. I1 and *I2* said that specifying conditions or hit counts is often "fuzzy (is it going to happen the 16th, 17th, or 18th time?)" and that once one knows the condition, one almost automatically understands the problem. Then, there is no need for the conditional breakpoint anymore. Moreover, "the types of problems where you need a conditional breakpoint happen very rarely" (*I2*). For example, when we presented the breakpoint export feature of Eclipse to *I2*, he replied "I did not know such a feature exists." Others said it is a "very esoteric thing" and that they have used it "maybe once or twice" (*I3*). This strengthens our intuition that debugging is an internal thought process not usually shared and that breakpoints are "like a one-shot. Ideally I wouldn't like them to be, but I just set them anew" (*I4*).

2) Debuggers are difficult to use. Another reason given by interviewees, even though seasoned engineers, was that "the debugger is a complicated beast" (*I2*) and that "debuggers that are available now are certainly not friendly tools and they don't lend toward self-exploration." Given our results on the use of features, we asked interviewees whether it might simply be enough to reduce the feature set. Both developers and *E1* to *E3* emphatically declined, arguing that "once you get into these crazy cases, they are really useful" (*I2*).

3) There is a lack of knowledge on how to use the debugger. When we asked developers where their knowledge of debugging comes from, many said that "big chunks are self-taught" and "[I] picked up various bits and pieces on the Internet" (*I4*). Even *I3*, the only interviewee who indicated that "debugging was explicitly covered [in my undergraduate]," said it is "partly self taught, partly [...] through key mentor ships." Making a case for hands-on teaching, he elaborated that "one of the engineers that mentored me [...] was some kind of wizard with GDB. I think when you meet someone who knows a very powerful tool it's very impressive and their speed to resolving something is much faster but it takes a lot of time to get to that point." Since we measured experience to have limited to no impact on (which) debugging features developers used (*SR6*), this hints at a lack of education on debugging that is pervasive from beginners and Computer Science students to experts. New Computer Science curricula that put debugging upfront could be an effective way to steer against it [358].

Time Effort for Debugging. Our study results *WR3* to *WR4* point to the fact that debugging in most cases is a short, "get-it-done" (*I1*) type of activity that, with only 14% of active IDE time (*WR3*) we found to consume significantly less than the 30–90% for testing and debugging reported by Beizer [359] at a general, project-level and the estimations by our interviewees, who gave a range of 20% to 60% of their active work time. One reason why our measured range is so much lower might be that developers (and humans in general) have a tendency to overestimate the duration of unpleasant tasks, as previously observed with testing [66]. Another is that developers included debugging tasks such as printf and the use of external tools, which we cannot measure. We need more studies to quantify this initial surprising finding. A common intuition in Software Engineering is that "small is better," since it is easier to manage and understand, see for example the recommendations to micro services, small commits, or short files. Contrary to this claim, we found that short classes need considerably more debugging (*WR11*) and that the longer amount of time developers spend in larger classes does not nearly compensate for it. Our interviewees agreed in unison that the hardest problems to debug are ones where interfaces or transactions between components are involved. Interfaces are typically short since they contain little logic, but represent a common source of integration problems and thus, the answers suggest, debugging effort. Then, longer classes are likely to have increased locality of features, which makes them often easier to understand [360] and thus probably also easier to troubleshoot. We need more research on this interaction between file length and debugging probability. Future studies could try to exploit the finding to recommend optimal system designs as a compromise between modularity and the ability to debug them.

Use of Tests for Debugging. In the survey, most respondents think (unit) testing is an

integral part of the debugging process, especially for reproducing bugs at the beginning of the process (*SR5, SR7*). However, there is mixed evidence on this in RQ V.2, as shown by *WR8, WR9* and *WR10*. On the one hand, failing tests do not seem to be a trigger for the start of debugging sessions. On the other hand, running tests in the IDE seems to be correlated with debugging more, while reading or modifying tests is not. Two factors can play a role: Developers who are more quality-concerned execute their tests more often and therefore also debug more. This is contrary to intuition and the answers of some of our interviewees, who claimed that as testing goes up, the debugging effort should decrease (*E2*): "Debugging is born of unknowns, and effective testing reduces these" (*I3*). An explanatory finding might be that the creation of tests itself adds code and complexity that might need to be debugged. We need more studies to research this interesting discovery.

Stepping Over the Point of Interest. We found that in less than about 5% of the debugging sessions the developer might have stepped over the point of interest and had to start debugging anew (*WR12*). This indicates that there is a limited, but existent gap in current debuggers process that might be filled by *back-in-time debuggers* [361]. Back-in-time debuggers allow developers to step back in the program execution in order to arrive at the point of interest without having to completely restart the debugging process. All our interviewees could relate to situations in which this occurred to them, stating that "it happens all the time" (*I1*) to "back in time debugger would be wonderful" (*I3*). However, *WR12* indicates that it might not be as frequent as some stated. While the drop frame feature allows developers to go to the beginning of the current method, it does not revoke side effects that already occurred and was therefore only found to be "helpful in a limited way" (*I3*). Currently, mainstream IDEs do not support back-in-time debugging.

Improvements in IDE Debuggers. We asked our interviewees how debugger creators could better support them. Their answers fall into two categories: 1) Make the core features easier to use while preserving all existing functionality. 2) Create tools that capture the holistic debugging process better. Elaborating on 2), *I1* denotes: "If you're in Java and have to debug across language boundaries, [...] you really get to a point where you feel helpless." Other wishes included the ability to do back-in-time debugging similar to Chronon [362], to have a live REPL, a feature the IDE xcode introduced [363].

> Developers have a strong need for back-in-time debuggers and an integrated REPL.

To improve the design of existing IDE debuggers with findings from our study, we arranged a meeting with three debugging project leads from Eclipse, *E1*, and an IDE developer from a commercial company, *E2*. The Eclipse leads said that, while they had sporadic evidence on how some individual developers use their debugger, they were unaware of the debugging behavior of a large population and the usage detail our study could provide. They started or updated six feature requests for the debugger based on our study, commencing work on bugs that had been dormant since 2004.[4] In the following, we focus on two already implemented features that are scheduled to roll out as part of Eclipse release 4.7.

[4]See umbrella bug 498469: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=498469`.

In our field study and interviews, we identified left-over breakpoints as a recurrent annoyance, which developers have to remove manually, with *I1* saying that suspending on old breakpoints unexpectedly interrupts his flow and that “every so often, once a week or so, I just delete all of them.” After making the Eclipse leads aware of this problem, they implemented age deprecation for breakpoints. It lets developers remove old breakpoints with one click. Although often referred to as a “dirty hack” (since it interferes with and pollutes production code), our study found that printf debugging also provides an advantage over the debugger’s watch view in that it preserves the history of past logs (for example, of memory entities in the watch view). Conversely, developers cannot enrich third-party libraries for which no source code is available with printf statements, but they can place debugger breakpoints in e.g. their Java byte code. To keep a history of logs when using the debugger, before our study, Eclipse and IntelliJ users had to set up an artificial construction of placing a conditional breakpoint that would print the information and always return false, thus never suspend. This hack of a “conditional breakpoint that is not conditional” (Bugtracker description) required intimate familiarity with the idiosyncrasies of the debugger and had bad performance, since code embedded in conditional breakpoints runs via the Java Debugging Infrastructure, which adds unnecessary overhead for a simple printout. By offering the new breakpoint type “tracepoint,” developers can now conveniently produce fast logs of debug traces. The Eclipse project implemented this simplified solution in Bug 71020, which had been in hibernation since 2004 and on which work commenced after our discussion.

6.5 Threats to Validity

In this section, we examine threats to the validity of our study and show how we mitigated them.

Construct Validity. The manual implementation of new functionality, such as the addition of the debug infrastructure to WatchDog, is prone to human errors. To minimize these risks, we extended WatchDog’s automated test suite. Furthermore, we use this test suite to make sure we introduced no regressions. In addition, we tested our plugins manually. Finally, we performed rigorous code reviews before we integrated the changes. Debug sessions might not correspond to actual debug work, e.g. a user might have inadvertently left the debugger in the IDE running, explaining our 90 hour outlier. However, such outliers are expected in an observational study of several months [64, 65]. Similarly, we approximate the number of classes in a project by the number of different classes we observe with WatchDog. Due to privacy reasons, we cannot mine the repositories of projects to gain an entirely correct figure.

Internal Validity. Since our survey in RQ V.1 dealt with debugging, participation might have been self-selecting, i.e. developers more interested and knowledgeable in debugging are more likely to have responded. We tried to contrast this with objective WatchDog observations, which is not advertised specifically as a debugging tool. An important internal threat is that the populations for RQ V.1 and RQ V.2 are different and their intersection is small (six users participated in both studies). However, we are confident we only encounter a small sampling or comparison bias because key characteristics of both populations are similar, as 1) 80% of respondents answered the survey for Java, which both

plugins work with in RQ V.2, 2) the majority in RQ V.1 used one of the IDEs supported in RQ V.2, 3) the experience distributions of both populations are similar and 4) both populations should be large enough to even out individual influences. Due to the fact that WatchDog gathers data automatically, it is harder for potentially evil-minded users to fabricate data than in surveys. Moreover, that the majority of data comes from a relatively small "power user" population (48 developers in our case, $top_{10\%}$ in *WR1*) is both normal in service use, for example on Twitter [364], and other observational studies [192, 365]. Discrepancies between some survey answers and the objective IDE observations have previously been observed in other studies [65].

External Validity. During our data collection period of more than two months we collected 1,155,189 intervals with a total duration of over ten developer years, spread over 458 users. The fact that over 80% of the survey respondents stem from the Java community means that little survey data is available about other communities. The same holds for the analysis of the WatchDog 2.0 data, which is restricted to the Java programming language and to the Eclipse and IntelliJ IDEs. Other IDEs are not included in our analysis and the results with them might deviate. However, at least imperative, statically typed languages similar to Java, such as C, C++, C#, or Objective-C, would likely yield similar results and are so widespread that researching them alone impacts many, if not the majority of, developers.

6.6 Conclusion

We set out to obtain a first cross-validated understanding of developers' debugging knowledge and contrasted it with their real-world IDE debugging behavior.

We found strong dichotomies in developers' opinions, knowledge, and behavior: Many believe modern debuggers to be superior to printf debugging, yet still employ it for many good reasons. IDE observations confirmed this finding, as only a third of developers ever invoked the debugger. We found that debugging is a technique defined by necessities: It is a relatively fast-paced and short-lived activity that is by nature so complicated that the tools around it should be as simple as possible. Consequently, developers use only basic features and seldom resort to more advanced breakpoint types or debugging techniques. Developers spend surprisingly little time in the debugger; only 13% of their total development time on average, in stark contrast to previous findings claiming more than 50%. As developers become more experienced, they seem to use the debugger slightly more, possibly because they educated themselves on its advanced affordances over printf debugging. We also found that having more tests in the code generally does not reduce the debugging burden, possibly because test code adds to the overall code that needs debugging.

In general, developers' theoretical knowledge and practical use of specialized debugging features are relatively shallow, just the amount that is seemingly sufficient for their debugging problems. Most developers said debugging was self-taught and not part of their curriculum. We believe that more educators can include practical, hands-on teaching, start in first year courses. Astonishingly, although bugs are inevitably linked with software and students learn programming in their introductory courses, they are only taught to properly debug much later, if ever.

Adding to this lack of debugging education, even experienced developers admitted

that debuggers are not easy to use. Apart from the wish for back-in-time debuggers, developers never expressed the wish for more debugging features. Instead of introducing ever more esoteric features, we therefore call to make using the already existing debugger features easier to use and more accessible. With the help of three Eclipse project leads, we identified several areas of improvement in the Eclipse debugger, leading to new simplified debugging features. One example is the introduction of a new breakpoint type in Eclipse that combines the advantages of debugger-instrumentation with the flexibility of printf debugging. Other IDE and debugger creators could follow this example and use our findings to further improve their debuggers.

7 Conclusion

This chapter revisits our original research questions, presents threats of validity concerning the thesis at large, contains concluding remarks on FDD and its implications, gives an overview of the contemporary scientific reception of the papers included in this thesis, and finishes with an outlook into future work on FDD.

7.1 Research Questions Revisited

In this section, we revisit and answer the research questions from Chapter 1.

RQ 1 How do developers use static analysis within FDD?

Our research shows that most state-of-the-art projects use at least one ASAT, but do not yet make use of the benefits of combining multiple ASATs. To make it easier for developers to see the benefits of combining ASATs on their projects, we invented the UAV prototype for Java. Moreover, by the example of the Last Line Effect, we have shown that even effects that we traditionally thought can only be revealed by manual code review, can be detected by an ASAT. An increasing number of ASATs and analyses enabled by them and a reluctance of developers to add new ASATs or update their configurations, call to increase automated notifications not maintained or setup by projects themselves, similar to the security vulnerabilities that GitHub has begun to send out. In particular, we examined this research question in two studies:

We first performed a qualitative and a quantitative studies on nine different ASATs for the programming languages Java, JavaScript, Ruby, and Python with populations of over 100 and 100,000 open-source projects respectively. Our results show that ASAT usage is widespread, but not ubiquitous, and that projects typically do not enforce a strict policy on ASAT use. Most ASAT configurations deviate slightly from the default, but hardly any introduce new custom analyses. Only a very small set of default ASAT analyses is widely changed. Finally, most ASAT configurations, once introduced, never change. If they do, the changes are small and have a tendency to occur within one day of the configuration's initial introduction. The results highlight developers' reluctance to introduce, change, or

remove an ASAT once officially introduced in the project. While this is good for consistency on the one hand, it might be an indication that the position of ASATs in most projects is not as central as they could be.

We then introduced and studied the “Last Line Effect,” the phenomenon that the last line or statement in a micro-clone is much more likely to contain an error than the previous lines or statements. We became aware of this phenomenon through manually reviewing source code first, and then used automated analyses that spotted so-called “micro-clones.” With their help, we analyzed 219 open source projects and reported on 263 faulty micro-clones. We interviewed six authors of these real-world faulty micro-clones to understand more about the circumstances under which they created them. We also examined the underlying mechanisms for the presence of these relatively trivial errors. Based on the interviews and further analyses, we suggest that, instead of technical reasons or the complexity of the task, so-called “action slips” play a pivotal role for the existence of the last line effect: Developers’ attention shifts away at the end of a micro-clone creation task due to noise and the routine nature of the task. All micro-clones whose origin we could determine were introduced in unusually large commits. Practitioners benefit from this knowledge twofold: 1) They can spot situations in which they are likely to introduce a faulty micro-clone and 2) they can use PVS-Studio, an automated micro-clone detector, to help find erroneous micro-clones. The implementation of a static check for Last Line-type errors shows that we can efficiently reduce the burden of code reviewers who would otherwise have to manually look for them. This indicates a possible rising importance for ASATs in the future, in particular in a FDD world.

RQ 2 How do developers use dynamic analysis within FDD?

7

Dynamic analysis within FDD comprises local and remote testing and debugging, see Figure 1.1. Similar to RQ 1, we find that developers have an ad-hoc mentality to using dynamic analysis. In particular, testing in the IDE is characterized by large gaps in which developers do not test, followed by relatively test-heavy development phases. Debugging is even more of a spontaneous activity, complemented by hard problems and a partly insufficient education on debugging. Moreover, testing did not seem to reduce the amount of debugging an individual developer had to do, possibly because the creation of tests leads to the discovery of further problems a developer has to debug. Debugging is driven by the desire to troubleshoot the problem at hand as quickly as possible and then (hopefully) forget about the experience. Subsequently, developers showed little interest to persist the results of their debugging sessions, for example through automatically generated tests. The more structured testing on the CI server is somewhat of an anti-pole to this and ensures that certain project rules and quality guidelines be followed. Nonetheless, developers also know how to “bend the rules” to use this concept to their advantage, for example to execute tests more efficiently than they locally could. We came to this conclusion by combining the results of three studies on dynamic analysis in FDD:

Our findings on developer testing in the IDE question several commonly shared assumptions and beliefs about developer testing: half of the developers in our study do not test at all. Developers rarely run their tests in the IDE, and consequently, most programming sessions in it end without them having executed any of their tests. Only once they start testing, do they do it extensively. A quarter of test cases run in the IDE are responsi-

ble for three quarters of all test failures. 12% of tests show non-deterministic behavior. Test-Driven Development (TDD) is not widely practiced. Finally, software developers only spend a quarter of their time engineering tests, whereas they think they test half of their time. Our initial theory on Test-Guided Development summarizes these practices of loosely guiding one's development efforts with the help of testing, a behavior we argue to be closer to the development reality of most developers than TDD.

In an analysis of over 2 million Java and Ruby builds on Travis CI, we found that testing is the single most important reason why builds fail. Moreover, the programming language had a strong influence on both the number of executed tests, their run time, and proneness to fail. The use of multiple integration environments lead to 10% more failures being caught at build time. However, testing on Travis CI does not seem to be an adequate surrogate for running tests locally in the IDE, as the time to feedback from the CI is typically two orders of magnitudes slower than in the IDE.

In a mixed-methods study, we found that IDE-provided debuggers are not used as often as expected, because "printf debugging" remains a feasible choice for many, a topic of heated discussion among programmers. Furthermore, both knowledge and use of advanced debugging features are low, further lowered by the complexity of most modern debuggers. Correlations on the individual developer level between the amount of time developers spent testing and debugging showed that there was no negative effect in our data. Thus, testing more did not seem to bare one from debugging, at least on the individual level. While debugging involves a lot of effort and often manually testing a piece of code, its only result is often a one liner code fix. Even so, developers showed little interest in trying to automatically persisting the fruits of their debugging efforts, for example by creating tests. Our results call for strengthening hands-on debugging experience in computer science curricula and have already refined the implementation of modern IDE debuggers.

7.2 Threats to Validity

In this section, we outline two general threats to validity that affect the conclusions of this thesis. These are of summarizing and complementary nature to the detailed threats for each study in their corresponding chapter.

- **External Validity**
  Generalizability and Incompleteness of Studied Systems. As we follow the observational path in the RPS (see Section 1.4.1), our case studies are naturally confined to a limited set of observations, tools, and environments. For example, in Chapter 2, we studied nine ASATs. A curated, but likely also incomplete list of static analysis tools on GitHub[1] alone lists 341 static analysis tools. Similar arguments can be made for WatchDog, which focuses on Java (GHTorrent lists 354 different programming languages on GitHub on April 1st, 2018) and TravisTorrent, which supports four programming languages out of the 26 supported by Travis CI (see Chapter 5). Moreover, in the WatchDog case, we only observe actions inside the IDE. We cannot measure actions that happen outside the IDE or when WatchDog is switched off.

[1] https://github.com/mre/awesome-static-analysis

We based our choice of studied tools and languages on several criteria: selecting tools that 1) have apparent practical relevance (for example, Java, FindBugs, or Eclipse), 2) represent the state-of-the-art in OSS for which we had some personal experience, and 3) which were likely to be generalizable outside their context (for example, Java could stand prototypical for most object-oriented languages). Strictly speaking, although we aimed for generality, the findings in this thesis are thus only true for the systems, the context, and the time period in which we studied them. However, we argue that it is likely we would find them replicable in other related contexts, as we did with our findings on automated developers tests. We first only studied developers in Eclipse and then could then replicate the findings in the IntelliJ and Visual Studio IDEs in Chapter 4.

- **Internal Validity**
  A large part of our studies rely on custom-made software (such as WatchDog and TravisTorrent) and bespoke data analyses written in R. In total, these comprise over 60,000 lines of code (see Table 1.3). Bugs in it could distort our results, as they have in the past (see Figure 4.12). Our research is not unique in this threat [366]. Where applicable, we covered our software with unit and integration tests and cross-validated the results in mixed-methods studies.

7.3 A Speculative Perspective on Feedback-Driven Development

In this section, we give a high-level compilation of our research findings, how they relate to each other and to the FDD model at large. We conclude with a more speculative outlook into the future of FDD.

7

By performing a history analysis on the configuration files of ASATs, we found that most of the over 100 top state-of-the-art projects on GitHub only use one ASAT. Moreover, this ASAT is typically only slightly customized and its customization normally does not evolve throughout a project's life time [60]. To help developers unleash the potential of multiple ASATs, we created the tool UAV [61]. On the intersection between manual code review and ASATs, we discovered the last line effect [62, 63], the startling realization that the last line or statement in a micro-clone is much more likely to contain a fault than any of the previous lines. Instead of technical, we singled out psychological reasons as the prime cause of the last line effect, most likely a working term memory overload due to the repetitive nature of the task. We also created ASAT rules to integrate the detection of faulty micro-clones in an automated FDD loop. Overall, our studies suggests that the integration of ASATs in the FDD cycle is currently still lacking behind the integration of dynamic analysis, particularly testing.

We found that the concept of "Test-Guided Development" best describes most developers' local testing practices [64–66], as they do not follow strict processes like TDD rigorously and tend to overestimate their testing efforts in the IDE twofold. With our family of WatchDog plugins [67], we studied the testing habits of more than 2,400 Java and C# developers in four different IDEs over the course of 2.5 years. Results suggest that testing practices largely generalize across (imperative) programming languages and IDEs. Most local testing is immediate, with a much shorter feedback loop than running the entire

test suite, which is usually offloaded to the CI. Accordingly, an analysis of our TravisTorrent "build log treasure trove" (Mathias Meyer, then-CEO of Travis CI [76]) shows that remote testing is the central phase of CI, causing more CI build failures than all other reasons combined [68]. Normally, contributions thus need to be reworked if they failed the testing stage of FDD ⑥ in figure 1.1. Finally, debugging is a somewhat opaque topic to developers not nearly as automated and streamlined as the other FDD quality assurance techniques: many developers still employ crude printf techniques, for lack of better knowledge or tools [70].

More generally, our investigation shows that the proposed FDD cycle is flexible enough to describe most contemporary software engineering projects, yet precise and distinct enough to distinguish itself clearly from past development methodologies like TDD that did not contain a tight integration of a multitude of feedback loops. While modern software development is principally characterized by its speed, we found several orders of magnitude of difference within the different feedback loops embedded in FDD: local testing is immediate, remote testing on the CI causes a notable delay, and waiting for a human reviewer to find time can further prolong the process. Reducing these delays, particularly in code review via the use of automated bots, would be one way to further increase development speed and robustness with regard to persons leaving a project.

On a higher level, a common denominator among our different study results is that instead of confining themselves on the methodologically correct usage of a method, developers constantly seem to seek ways and *shortcuts* in which they can use the method to their advantage. This way, they might achieve higher development speed in the FDD cycle, for example: instead of blocking their machine for 10 minutes to execute all UI tests, developers in the Eclipse project stated that they phase out these UI tests to the CI via their Gerrit code review system. Having not completed their local test-feedback loop yet, they did this knowing full well that the contribution was not ready for code review. Another instance of this is the dedicated use of "printf" statements to debug issues, which most developers agree to be far from ideal in a methodological sense. However, for many practical purposes, it seems to *get the job done* in the most efficient way.

Similarly, developers perform a test-guided development approach with test-heavy development sessions as they see fit. FDD can thus also capture this *hacker's mindset* to software development, which has recently been propagated at companies such as Facebook [367]. A programmer consciously makes a decision to deviate from a norm to further one's direct goal, as far as the FDD tools employed by the project allow. In some sense, this is a trend to go back to the roots of computer programming, leaving processes and methodologies behind that seemed to hinder progress, while still making use of recently developed tools such as CI and pull-based development. Researchers need to take this cleverness of the developers they study into account. Developers should be aware of the fallacies that skipping corners brings with it, which bears the risk of manifesting itself in bugs visible to the user. A relatively rigid, highly automated FDD process with some room for flexibility on an individual contributor's side could thus be both a driver of creativity ("form liberates") and act as a safety net against too much corner cutting.

7.4 Implications

In this section, we give an outline of how scientists and developers could and in part have already used the work in this thesis to further the development of Software Engineering research and tools.

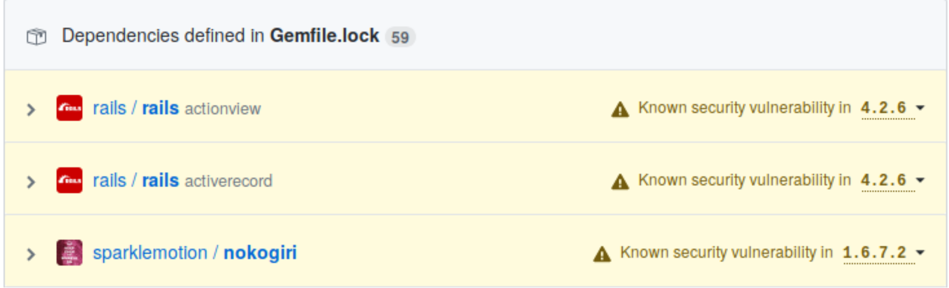


Figure 7.1: GitHub displaying push-based security vulnerabilities to a project maintainer.

The FDD model gives us a framework that can steer our thinking of how programmers drive software development by defining a vocabulary that allows us to efficiently communicate about modern-day collaborative software development. In this thesis, we have (1) introduced the notion of FDD and (2) defined and empirically characterized its sub-components. Scientists can further refine this preliminary model empirically, for example by augmenting our studies on ASATs (Chapter 2) and remote testing on the CI (Chapter 5) with data from closed-source projects. This would align the study population with the studies on local testing and debugging, which already comprise a mix of OSS and closed source data. Researchers could further study and compare different instantiations of the FDD model across projects and so ultimately come up with recommended best practices by identifying which parts and particular implementations of the FDD are useful to achieve better software quality in practice.

7

7.4.1 Individual FDD Stages

In November 2017, GitHub started to send developers notifications about potential security problems in their projects like the ones shown in Figure 7.1 [368]. These push-based notifications are different from the pull-based feedback that developers receive from setting up their own ASATs – GitHub analyzes all repositories for possible vulnerabilities without further actions by the project owners. It then automatically notifies the project owners if one is found. This has the advantage that projects do not need to configure an additional ASAT, which developers are reluctant to do, as we have shown in Chapter 2. It would be interesting to study whether this approach leads to more-widespread fixes of these security vulnerabilities than would otherwise be the case. However, in FDD, developers are also concerned with a high number of false positive warnings (Chapter 2). To reduce these, we need to define more precise methods than just checking on the package-level whether projects are susceptible to a certain vulnerability of a dependency feature they might not even be using. While our investigation focused on a largely quantitative analysis of software artifacts, Vassallo et al. took a developer-centric stance on ASATs

and re-used our GDC [369]. Their survey of 41 developers confirmed core findings of our study, for example that developers tend to only evolve ASAT configurations hesitantly.

In Chapter 3, we introduced the Last Line Effect. Based on our definition and algorithms to identify micro-clones, van Tonder implemented checks for faulty micro-clones in Facebook's ASAT for PHP, PFFF. Van Tonder and Le Goues then performed a study on the prevalence of such micro-clones among more than 380,000 Java repositories on GitHub and found more than 24,000 faulty micro-clones in them. They provided fixes to projects for 43 of these, directly having an impact on the quality of OSS projects. While technically challenging to detect, researchers have not nearly exhausted the research possibilities that the definition of micro-clones has opened up.

Chapter 5 introduced the concept of Continuous Integration build log analysis. We used it to gain insights into testing in a CI environment. The next logical step was to apply the same approach to study ASATs use, as discussed in Chapter 2, in a CI environment, which Zampetti et al. followed up with [370]. Rausch et al. also used and described the two underlying technologies we devised to study Travis CI, namely an automatic buildlog parsing and a method to link Travis builds to Git commits on GitHub [371]. Chapter 5 also suggested a method to discern flaky tests by triggering a re-execution of their builds on Travis CI. Labuschagne et al. used this method to determine the flakiness of tests on Travis CI [372].

With our study on Travis CI (Chapter 5), we have done research into the CI server. Our study was confined to the Ruby and Java ecosystems. We need a comparison of CI practices across a wider-body of languages, particularly from other programming paradigms such as functional languages, to gain a holistic understanding of CI. TravisTorrent has seen adoption in- and outside the MSR community (Section 7.4). The TravisTorrent data set has already allowed fellow researchers to predict the outcome of a build or its duration with very high accuracy [373, 374], measure the difference of external versus internal contributions [375], determine the influence of the sentiment expressed in a commit message on its build outcome [376], do a general factor analyses to discover the components influencing CI build outcomes [377, 378], or perform time-dependent analyses [379]. At this point, TravisTorrent is a static snapshot of a data set over a fixed number of projects that requires considerable manual efforts to update. A live version of TravisTorrent with streaming analytics would allow not only researchers to obtain information from other than our pre-selected projects, but also give project maintainers and developers the option to retrieve build statistics about their own projects. Coupled with highly accurate build prediction (Section 7.4), the use of such data could significantly shorten the feedback cycle from remote testing. Practitioners do not necessarily have to resort to complex machine learning algorithms to gain advantages from it: Having the CI environment order test executions based on historic run information could be an easy and efficient way to make builds fail fast without having to build a prediction model.

With WatchDog, we studied how dynamic analysis works locally (Chapter 4). Similarly, researchers could re-purpose WatchDog to study how developers work with feedback from ASATs in their IDEs. In fact, as an infrastructure, WatchDog lends itself to generalization studies of any phenomenon about development work a researcher might be interested in. As such, scientists could use it to study for example the exact code creation process ② in Figure 1.1, which is outside the scope of FDD. Local testing (or ASAT

use), however, is not intrinsically refined to the IDE. Other places where it happens is on the command-line or the CI server. Similarly, the primary environment of our study on debugging (Chapter 6) was the IDE, but debugging can happen in a myriad of environments, for example in distributed systems. We need knowledge on how prevalent the use of each of the different environments is and how developers' usage patterns varies across each other. Closely replicating some of our research questions on developer testing in the IDE (Chapter 4), Blondeau et al. studied the testing habits of developers in one large IT company [380]. Using their purpose-built IDE plugins for IntelliJ and Elcipse, they were able to quantify that the testing behavior of the developers at the company in some cases outperformed our participants, and in other cases very closely confirmed our observations.

7.4.2 Conclusion and Future Work on FDD

One could argue that channels such as the Q&A site Stack Overflow form a feedback-loop, too, and characterize them in an extended version of FDD. Channels like chats such as IRC, Q&A sites, fora, and mailing lists also differ in substantial aspects from code reviewing. One key aspect is that developers can utilize crowd knowledge from programmers outside their own organization. In contrast to code review, however, participation in such fora is typically completely voluntary. These qualities make such fora unusual and interesting to study, since we do not fully understand their impact on development and their own dynamics.

In this thesis, we also studied the relationship between different components of FDD. For example, intuitively, one would assume that spending more effort on testing would at least eventually reduce developers' debugging burden. However, at least at the individual contributor level, that did not seem to be the case. Instead, it might have other benefits, by pushing the discovery of problems upfront, rather than when the software is already running in production. Moreover, we do not know how the presence of tests in the project overall influences debugging, because we did not have insight into how many tests existed in the project in total. We need more holistic research also capturing a project's repository to understand this relationship better. Generally, the implementation of any stage in the FDD process should lead to higher developer productivity or more efficient discovery of problems, a promising avenue for future work.

Our FDD model describes and focuses on how developers interact with quality assurance methods that generate feedback. It is not, however, a complete theory of how software development works. For example, interactions with other stakeholders, projects, or the entire programming ecosystem are outside its current scope. Following Sjøberg et al. [381], to build a theory from the FDD model requires us to follow a principled structured research approach, which would holistically address and validate the FDD model. Going further, researchers could create a comprehensive theory of modern day Software Engineering and embed FDD as one of its components.

Bibliography

URLs in this thesis have been archived on Archive.org. Their link target in digital editions refers to this timestamped version.

References

[1] Moritz Beller. Toward an empirical theory of feedback-driven development. In *40th International Conference on Software Engineering, ICSE 2018, Student Research Competition, Gothenborg, Sweden*, 2018. Open Access version: `https://pure.tudelft.nl/portal/files/40152814`.

[2] Kevin Forsberg and Harold Mooz. The relationship of systems engineering to the project cycle. *Engineering Management Journal*, 4(3):36–43, 1992.

[3] Martin Fowler and Jim Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[4] LBS Raccoon. The chaos model and the chaos cycle. *ACM SIGSOFT Software Engineering Notes*, 20(1):55–66, 1995.

[5] Kent Beck. *Test Driven Development – by Example.* Addison Wesley, 2003.

[6] Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387. IEEE, 2011.

[7] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 378–389. ACM, 2014.

[8] Diomidis Spinellis. Reading, writing, and code. *Queue*, 1(7):84, 2003.

[9] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[10] Jean Hartmann and David J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31–36, 1990.

[11] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques.* Wiley Publishing, 1998.

[12] Laura Lehtola, Marjo Kauppinen, and Sari Kujala. Requirements prioritization challenges in practice. *Product focused software process improvement*, pages 497–508, 2004.

[13] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[14] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. Continuous delivery practices in a large financial organization. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 519–528, 2016. Open Access version: `https://pure.tudelft.nl/portal/files/9159936/vassalloICSME2016.pdf`.

[15] Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, et al. The top 10 adages in continuous deployment. *IEEE Software*, 34(3):86–95, 2017.

[16] Katja Kevic, Brendan Murphy, Laurie Williams, and Jennifer Beckmann. Characterizing experimentation in continuous deployment: a case study on bing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 123–132. IEEE Press, 2017.

[17] Selman Ercan, Quinten Stokkink, and Alberto Bacchelli. Predicting answering times on stack overflow. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 442–445. IEEE Press, 2015.

[18] Kenneth Orr. Stack overflow is pretty amazing, 2010. `https://explodingpixels.wordpress.com/2010/01/08/stack-overflow-is-pretty-amazing/`. Accessed January 19, 2018.

[19] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE Press, 2013.

[20] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Social computing (SocialCom), 2013 international conference on*, pages 188–195. IEEE, 2013.

[21] Ashton Anderson, Daniel Huttenlocher, Jon Kleinberg, and Jure Leskovec. Discovering value from community activity on focused question answering sites: a case study of stack overflow. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 850–858. ACM, 2012.

[22] Vaclav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE software*, 21(4):62–69, 2004.

[23] BA Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, and DWR Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2):69–75, 1995.

[24] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[25] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[26] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[27] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 202–211. ACM, 2014.

[28] Brittany Itelia Johnson et al. A tool (mis) communication theory and adaptive approach for supporting developer tool use, 2017. PhD Thesis.

[29] C. Lebeuf, M. A. Storey, and A. Zagalsky. Software bots. *IEEE Software*, 35(1):18–23, January 2018.

[30] Josh Lerner and Jean Tirole. Some simple economics of open source. *The journal of industrial economics*, 50(2):197–234, 2002.

[31] A Guzzi. *Supporting Developers' Teamwork from within the IDE.* PhD thesis, TU Delft, Delft University of Technology, 2015.

[32] Jim Whitehead, Ivan Mistrík, John Grundy, and André Van der Hoek. Collaborative software engineering: concepts and techniques. In *Collaborative Software Engineering*, pages 1–30. Springer, 2010.

[33] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE, May 2016.

[34] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 358–368. IEEE, 2015.

[35] Vinod Valloppillil. Open source software: A (new?) development methodology. *Internal Microsoft memo*, 1998. http://www.catb.org/esr/halloween/halloween1.html. Accessed November 29, 2017.

[36] Microsoft Corp. Microsoft open source programs office - providing open source community tooling, guidance and playbooks, 2017. https://opensource.microsoft.com/resources. Accessed November 29, 2017.

[37] Steven Vaughan-Nichols. Why microsoft is turning into an open-source company, 2017. http://www.zdnet.com/article/why-microsoft-is-turning-into-an-open-source-company/. Accessed November 29, 2017.

[38] Amanda Lee, Jeffrey C Carver, and Amiangshu Bosu. Understanding the impressions, motivations, and barriers of one time code contributors to floss projects: a survey. In *Proceedings of the 39th International Conference on Software Engineering*, pages 187–197. IEEE Press, 2017.

[39] Nadia Eghbal. Roads and bridges: The unseen labor behind our digital infrastructure. *Ford Foundation*, 2016.

[40] Nolan Lawson. What it feels like to be an open-source maintainer, 2017. https://nolanlawson.com/2017/03/05/what-it-feels-like-to-be-an-open-source-maintainer/. Accessed November 29, 2017.

[41] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 233–236. IEEE, 2013.

[42] Barry Boehm, Hans Dieter Rombach, and Marvin V Zelkowitz. *Foundations of empirical software engineering: the legacy of Victor R. Basili*. Springer Science & Business Media, 2005.

[43] Laurent Bossavit. *The Leprechauns of Software Engineering*. Lulu.com, 2015.

[44] Mary Shaw. Writing good software engineering research papers: minitutorial. In *Proceedings of the 25th international conference on software engineering*, pages 726–736. IEEE Computer Society, 2003.

[45] B. Kitchenham and S Charters. Guidelines for performing systematic literature reviews in software engineering, 2007.

[46] Antonia Bertolino. The (im)maturity level of software testing. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, September 2004.

[47] Mika V Mäntylä, Juha Itkonen, and Joonas Iivonen. Who tested my software? testing as an organizationally cross-cutting activity. *Software Quality Journal*, 20(1):145–172, 2012.

[48] E Mcgrath. Methodology matters: Doing research in the behavioral and social sciences. In *Readings in Human-Computer Interaction: Toward the Year 2000 (2nd ed.* Citeseer, 1995.

[49] Klaas-Jan Stol and Brian Fitzgerald. Theory-oriented software engineering. *Science of Computer Programming*, 101:79–98, 2015.

[50] Foster Provost and Tom Fawcett. Data science and its relationship to big data and data-driven decision making. *Big Data*, 1(1):51–59, 2013.

[51] John G Adair. The Hawthorne effect: A reconsideration of the methodological artifact. *Journal of applied psychology*, 69(2):334–345, 1984.

[52] David A Braunholtz, Sarah JL Edwards, and Richard J Lilford. Are randomized clinical trials good for us (in the short term)? evidence for a "trial effect". *Journal of clinical epidemiology*, 54(3):217–224, 2001.

[53] Will G Hopkins. *A new view of statistics*. 1997. `http://sportsci.org/resource/stats/`. Accessed March 27, 2017.

[54] Kathy Charmaz and Linda Liska Belgrave. *Grounded theory*. Wiley Online Library, 2007.

[55] Adil E Shamoo and David B Resnik. *Responsible conduct of research*. Oxford University Press, 2009.

[56] Wikipedia. Open science, 2018. `https://en.wikipedia.org/wiki/Open_sciencex`. Accessed February 6th, 2018.

[57] NWO. Open science, 2018. `https://www.nwo.nl/en/policies/open+science`. Accessed February 6th, 2018.

[58] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The FAIR guiding principles for scientific data management and stewardship. *Scientific data*, 3, 2016.

[59] Creative Commons. About the licenses. 2018. `https://creativecommons.org/licenses/`. Accessed February 7th, 2018.

[60] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 470–481. IEEE, 2016. Open Access version: `https://pure.tudelft.nl/portal/files/8928493/2016_beller_bholanath_mcintosh_zaidman_analyzing_the_state_of_static_analysis_a_large_scale_evaluation_in_open_source_software.pdf`.

[61] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong andg Sunwei Wang, Moritz Beller, and Andy Zaidman. UAV: warnings from multiple automated static analysis tools at a glance. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 472–476, 2017. Open Access version: https://pure.tudelft.nl/portal/files/32869492/buckersSANER2017_2.pdf.

[62] Moritz Beller, Andy Zaidman, Andrey Karpov, and Rolf A. Zwaan. The last line effect explained. *Empirical Software Engineering*, 22(3):1508–1536, Jun 2017. Open Access version: https://pure.tudelft.nl/portal/files/13457316/llee.pdf or https://link.springer.com/article/10.1007%2Fs10664-016-9489-6 (publisher).

[63] Moritz Beller, Andy Zaidman, and Andrey Karpov. The last line effect. In *23rd International Conference on Program Comprehension (ICPC)*, pages 240–243. ACM, 2015. Open Access version: https://pure.tudelft.nl/portal/files/8957994/8928291.pdf.

[64] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017. To appear. Pre-print: http://ieeexplore.ieee.org/document/8116886/.

[65] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *Proceedings of the 37th International Conference on Software Engineering (ICSE), NIER Track*, pages 559–562. IEEE, 2015. Open Access version: https://pure.tudelft.nl/portal/files/8928078/2015_beller_gousios_zaidman_how_much_do_developers_test.pdf.

[66] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 179–190. ACM, 2015.

[67] Moritz Beller, Igor Levaja, Annibale Panichella, Georgios Gousios, and Andy Zaidman. How to catch 'em all: Watchdog, a family of ide plug-ins to assess testing. In *3rd International Workshop on Software Engineering Research and Industrial Practice (SER&IP 2016)*, pages 53–56. IEEE, 2016. Open Access version: https://pure.tudelft.nl/portal/files/8928027/2016_beller_levaja_panichella_gousios_zaidman_how_to_catch_em_all_watchdog_a_family_of_ide_plug_ins_to_assess_testing.pdf.

[68] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, 2017. Open Access version: https://pure.tudelft.nl/portal/files/21809641/bellerMSR2017.pdf.

[69] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings*

of the Proceedings of the 14th International Conference on Mining Software Repositories (MSR), 2017. Open Access version: https://pure.tudelft.nl/portal/files/21809391/bellerMSR2017miningchallenge.pdf.

[70] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *40th International Conference on Software Engineering, ICSE 2018, Gothenborg, Sweden*, 2018. Open Access version: https://pure.tudelft.nl/portal/files/38319543/paper.pdf.

[71] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: an empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 547–558, 2016. Open Access version: https://pure.tudelft.nl/portal/files/8927923/PID4080971.pdf.

[72] Alberto Bacchelli and Moritz Beller. Double-blind review in software engineering venues: the community's perspective. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 385–396, 2017. Open Access version: https://pure.tudelft.nl/portal/files/17721365/1589a385.pdf.

[73] Moritz Beller. Wie viel testen ist genug? *Eclipse Magazin 1.15, Frankfurt am Main (Germany)*, 2014.

[74] Moritz Beller. Test oder nichttesten, das ist hier die frage. *Eclipse Magazin 2.16, Frankfurt am Main (Germany)*, 2016.

[75] Moritz Beller. What we learned from analyzing 2+ million travis builds. A summary of the paper [68] in the Travis CI blog. https://blog.travis-ci.com/2016-07-28-what-we-learned-from-analyzing-2-million-travis-builds/.

[76] Moritz Beller. Become a travis ci log miner in the msr mining challenge 2017! https://blog.travis-ci.com/2017-01-16-travis-ci-mining-challenge/.

[77] Moritz Beller, Andy Zaidman, Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, and Sunwei Wang. The unified asat visualizer (uav), a tool for comparing multiple asats on your java projects. *IEEE Software Blog*. http://blog.ieeesoftware.org/2017_01_01_archive.html.

[78] NASA. JPL C Standard, 2015. Accessed on: November 14th, 2015.

[79] NASA. JPL Java Standard, 2015. Accessed on: November 14th, 2015.

[80] S. C. Johnson. Lint, a c program checker. In *Computer Science Technical Report 65*. Bell Laboratories, 1977.

[81] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[82] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[83] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217(0):5–21, 2008.

[84] Coverity Inc. Effective management of static analysis vulnerabilities and defects. White paper, Coverity Inc., 2009.

[85] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011.

[86] Joseph Ruthruff, John Penix, David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, pages 341–350. ACM, 2008.

[87] Jasper Kamperman. Automated software inspection: A new approach to increased software quality and productivity. White paper, Reasoning Inc., 2002.

[88] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50. ACM, 2008.

[89] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[90] Nick Rutar, Christian Almazan, and Jeffrey Foster. A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering*, pages 245–256, 2004.

[91] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. *Comparing Bug Finding Tools with Reviews and Tests*, volume 3502 of *Lecture Notes in Computer Science*, book section 4, pages 40–55. Springer Berlin Heidelberg, 2005.

[92] Fadi Wedyan, Dalal Alrmuny, and James Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *International Conference on Software Testing Verification and Validation*, pages 141–150. IEEE, 2009.

[93] Sarah Heckman. Adaptively ranking alerts generated from automated static analysis. *Crossroads*, 14(1):1–11, 2007.

[94] Sunghun Kim and Michael Ernst. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54. ACM, 2007.

[95] Ted Kremenek and Dawson Engler. *Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations*, volume 2694 of *Lecture Notes in Computer Science*, book section 16, pages 295–315. Springer Berlin Heidelberg, 2003.

[96] Nathaniel Ayewah, William Pugh, David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[97] Stefan Wagner, Florian Deissenboeck, Michael Aichner, Johann Wimmer, and Markus Schwalb. An evaluation of two bug pattern tools for java. In *1st International Conference on Software Testing, Verification, and Validation*, pages 248–257, 2008.

[98] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.

[99] Cesar Couto, João Montandon, Christofer Silva, and Marco Tulio Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, 21(2):241–257, 2013.

[100] Akash Kumar Tripathi and Atul Gupta. A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for java programs. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 23. ACM, 2014.

[101] IEEE. Ieee standard classification for software anomalies. *IEEE Std 1044-1993*, pages 1–32, 1994.

[102] Ram Chillarege, Inderpal Bhandari, Jarir Chaar, Michael Halliday, Diane Moebus, Bonnie Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

[103] Nachiappan Nagappan, Laurie Williams, John Hudepohl, Will Snipes, and Mladen Vouk. Preliminary results on using static analysis tools for software inspection. In *15th International Symposium on Software Reliability Engineering*, pages 429–439, 2004.

[104] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John Hudepohl, and Mladen Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.

[105] Mika Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[106] Khaled El Emam and Isabella Wieczorek. The repeatability of code defect classifications. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 322–333, 1998.

[107] Jay P Kesan and Rajiv C Shah. Setting software defaults: Perspectives from law, computer science and behavioral economics. *Notre Dame L. Rev.*, 82:583, 2006.

[108] GitLab Inc. Code, test, and deploy together, 2015. https://about.gitlab.com/about/. Accessed November 14, 2015.

[109] Google Inc. Bidding farewell to google code, 2015. http://google-opensource.blogspot.nl/2015/03/farewell-to-google-code.html. Accessed November 6, 2017.

[110] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 384–387. ACM, 2014.

[111] R. Bholanath. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. Master's thesis, Delft University of Technology, 2015. http://repository.tudelft.nl/view/ir/uuid:3d834130-8dd7-420a-9af9-6e77761cdad6/.

[112] The Abilian Team. Abilian github repository, 2015. Accessed on: November 14th, 2015.

[113] GNU. The bash shell, 2015. https://gnu.org/software/bash/bash.html. Accessed November 14th, 2015.

[114] Coverity Inc. Software testing and static analysis tools | coverity, 2014. http://www.coverity.com/. Accessed October 3, 2014.

[115] Stack Exchange Inc. Stack Overflow, 2015. http://stackoverflow.com/. Accessed November 9, 2017.

[116] Oliver Burn. checkstyle - checkstyle 5.9-snapshot, 2017. http://checkstyle.sourceforge.net/. Accessed October 14, 2017.

[117] FindBugs. Findbugs™ - find bugs in java programs, 2014. http://findbugs.sourceforge.net/. Accessed October 2, 2015.

[118] PMD. Pmd, 2014. http://pmd.sourceforge.net/. Accessed November 7, 2017.

[119] ESLint. Eslint - pluggable javascript linter, 2015. http://eslint.org/. Accessed November 7, 2015.

[120] JSCS. Jscs - about, 2015. http://jscs.info/. Accessed May 7th, 2015.

[121] Anton Kovalyov. Jshint, a javascript code quality tool, 2015. http://jshint.com/. Accessed November 7, 2015.

[122] Matthias Miller. Javascript lint, 2015. Accessed on: November 7th, 2015.

[123] Logilab. Pylint - code analysis for python | www.pylint.org, 2015. http://pylint.org/. Accessed May 7, 2015.

[124] Bozhidar Batsov. Rubocop | a ruby static code analyzer, 2015. http://batsov.com/rubocop/. Accessed November 7, 2017.

[125] Anton Kovalyov. http://jshint.com/docs/options/. jshint option reference, 2015. Accessed May 8, 2015.

[126] Aragon Consulting Group, Inc. Krugle – #1 for enterprise code search, 2015. http://www.krugle.com. Accessed November 14, 2015.

[127] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph Johnson, and Danny Dig. *Is it dangerous to use version control histories to study source code evolution?*, pages 79–103. ECOOP 2012–Object-Oriented Programming. Springer, 2012.

[128] Rahul Kumar and Aditya Nori. The economics of static analysis tools. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 707–710. ACM, 2013.

[129] Georgios Gousios, Andy Zaidman, Margaret-Anne D. Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 358–368. IEEE, 2015.

[130] Travis CI GmbH. Travis continuous integration, 2015. Accessed on: November 14th, 2015.

[131] Coverity Inc. Coverity scan - github integration, 2015. https://scan.coverity.com/github. Accessed October 21, 2015.

[132] Coverity Inc. Coverity scan - travis ci integration, 2017. https://scan.coverity.com/travis_ci. Accessed November 6, 2017.

[133] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *Proc. International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 161–170. IEEE, 2015.

[134] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *Proc. International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166. IEEE, 2016.

[135] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proc. of the 2nd Conference on Visualization (VIS)*, pages 284–291. IEEE, 1991.

[136] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, 2015.

[137] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. Teamscale: Software quality control in real-time. In *Companion Proceedings of the Int'l Conference on Software Engineering (ICSE)*, pages 592–595. ACM, 2014.

[138] G Campbell and Patroklos P Papapetrou. *SonarQube in Action*. Manning Publications Co., 2013.

[139] Coverity Scan Static Analysis. https://scan.coverity.com/. Accessed November 6, 2017.

[140] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. Online Appendix for UAV: Warnings From Multiple Automated Static Tools At A Glance. https://figshare.com/s/05658ac8ff03d57a8d60.

[141] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering*, pages 161–170. IEEE, 2015.

[142] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. Teamscale: Software quality control in real-time. In *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE'14)*, 2014.

[143] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proc. International Symposium on Empirical Software Engineering (ISESE)*, pages 83–92. IEEE, 2004.

[144] Cory J Kapser and Michael W Godfrey. Cloning considered harmful–considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[145] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[146] Chanchal Roy, James Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[147] J Martin Bland and Douglas G Altman. The odds ratio. *Bmj*, 320(7247):1468, 2000.

[148] Rainer Koschke. Survey of research on software clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[149] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 485–495. IEEE, 2009.

[150] John R Anderson. *Cognitive psychology and its implications*. WH Freeman/Times Books/Henry Holt & Co, 1990.

[151] M. Botvinick and D. C. Plaut. Doing without schema hierarchies: A recurrent connectionist approach to routine sequential action and its pathologies. *Psychological Review*, 111:395–429, 2004.

[152] R.P. Cooper and T. Shallice. Hierarchical schemas and goals in the control of sequential behaviour. *Psychological Review*, 113:887–916, 2006.

[153] J. G. Trafton, E. M. Altmann, and R. M. Ratwani. A memory for goals model of sequence errors. *Cognitive Systems Research*, 12:134–143, 2011.

[154] Michael J Kane, Leslie H Brown, Jennifer C McVay, Paul J Silvia, Inez Myin-Germeys, and Thomas R Kwapil. For whom the mind wanders, and when an experience-sampling study of working memory and executive control in daily life. *Psychological science*, 18(7):614–621, 2007.

[155] Alice F Healy. Proofreading errors on the word the: New evidence on reading units. *Journal of Experimental Psychology: Human Perception and Performance*, 6(1):45, 1980.

[156] Julia E Moravcsik and Alice F Healy. Effect of meaning on letter detection. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 21(1):82, 1995.

[157] JOHN J O'Malley and John Gallas. Noise and attention span. *Perceptual and motor skills*, 44(3):919–922, 1977.

[158] Keisuke Fukuda and Edward K Vogel. Human variation in overriding attentional capture. *The Journal of Neuroscience*, 29(27):8726–8733, 2009.

[159] Rijnard van Tonder and Claire Le Goues. Defending against the attack of the microclones. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–4. IEEE, 2016.

[160] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 255–265. ACM, 2015.

[161] Michiel de Wit, Andy Zaidman, and Arie van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 169–178. IEEE, 2009.

[162] Leo Meyerovich and Ariel Rabkin. Empirical analysis of programming language adoption. In *ACM SIGPLAN Notices*, volume 48, pages 1–18. ACM, 2013.

[163] Chanchal K. Roy, Minhaz F. Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, (CSMR-WCRE)*, pages 18–33. IEEE, 2014.

[164] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377. IEEE, 1998.

[165] Hamid Abdul Basit and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 513–516. ACM, 2007.

[166] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the International Software Metrics Symposium (METRICS)*, pages 292–303. IEEE, 1999.

[167] Cory Kapser and Michael Godfrey. A taxonomy of clones in source code: The reengineers most wanted list. In *2nd International Workshop on Detection of Software Clones (IWDSC-03)*, volume 13, 2003.

[168] Jeffrey Svajlenko and Chanchal Kumar Roy. Evaluating modern clone detection tools. In *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 321–330. IEEE, 2014.

[169] Minhaz F. Zibran, Ripon K. Saha, Muhammad Asaduzzaman, and Chanchal K. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 295–304. IEEE, 2011.

[170] Chanchal K. Roy and James R. Cordy. A survey on software clone detection research. Technical Report TR 2007-541, Queens University, 2007.

[171] Matthias Rieger, Stephane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 100–109. IEEE, 2004.

[172] Debarshi Chatterji, Jeffrey C Carver, Beverly Massengil, Jason Oslin, Nicholas Kraft, et al. Measuring the efficacy of code clone information in a bug localization task: An empirical study. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 20–29. IEEE, 2011.

[173] Nils Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, 2011.

[174] Katsuro Inoue, Yoshiki Higo, Norihiro Yoshida, Eunjong Choi, Shinji Kusumoto, Kyonghwan Kim, Wonjin Park, and Eunha Lee. Experience of finding inconsistently-changed bugs in code clones of mobile software. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 94–95. IEEE, 2012.

[175] Shuai Xie, Foutse Khomh, and Ying Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013.

[176] Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[177] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 12–13. ACM, 2014.

[178] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 33:1–33:11. ACM, 2012.

[179] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[180] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of the International Conference on Software Engineering (ISCE), Workshop on the Future of Software Engineering (FOSE)*, pages 85–103, 2007.

[181] Frederick Brooks. *The mythical man-month.* Addison-Wesley, 1975.

[182] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220–229. IEEE, 2008.

[183] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples.* Wiley, 2012.

[184] Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. Enriching in-ide process information with fine-grained source code history. In *Proceedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering*, pages 250–260. IEEE, 2017.

[185] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, 2007.

[186] Robert L. Glass, Ross Collard, Antonia Bertolino, James Bach, and Cem Kaner. Software testing and industry needs. *IEEE Software*, 23(4):55–57, 2006.

[187] John Rooksby, Mark Rouncefield, and Ian Sommerville. Testing in the wild: The social and organisational dimensions of real world practice. *Comput. Supported Coop. Work*, 18(5-6):559–580, December 2009.

[188] TestRoots WatchDog. `https://github.com/TestRoots/watchdog`. Accessed June 6, 2017.

[189] Paul Muntean, Claudia Eckert, and Andreas Ibing. Context-sensitive detection of information exposure bugs with symbolic execution. In *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices (InnoSWDev)*, pages 84–93. ACM, 2014.

[190] Google Inc. and the Open Handset Alliance. Download android studio and sdk tools. Accessed 2017/05/31.

[191] ReSharper Plugin Gallery. `https://www.jetbrains.com/resharper/`. Accessed June 6, 2017.

[192] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A study of visual studio usage in practice. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134. IEEE, 2016.

[193] Moritz Beller, Niels Spruit, and Andy Zaidman. How developers debug. *PeerJ Preprints*, 5:e2743v1, 2017.

[194] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[195] Apache Maven Conventions. `http://maven.apache.org/guides/getting-started`. Accessed June 6, 2017.

[196] John C. Munson and Sebastian G. Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, page 24. IEEE, 1998.

[197] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[198] Hussan Munir, Krzysztof Wnuk, Kai Petersen, and Misagh Moayyed. An experimental evaluation of test driven development vs. test-last development with industry professionals. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 50:1–50:10. ACM, 2014.

[199] Yahya Rafique and Vojislav B. Misic. The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, 39(6):835–856, 2013.

[200] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition).* Pearson Higher Education, 2004.

[201] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata theory, languages, and computation.* Prentice Hall, 2007.

[202] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[203] J. L. Devore and N. Farnum. *Applied Statistics for Engineers and Scientists.* Duxbury, 1999.

[204] Sven Amann, Sebastian Proksch, and Sarah Nadi. FeedBaG: An Interaction Tracker for Visual Studio. In *Proceedings of the 24th International Conference on Program Comprehension*, pages 1–3. IEEE, 2016.

[205] Let's Develop With TestRoots' WatchDog. `http://youtu.be/-06ymo7dSHk`. Accessed June 7, 2017.

[206] Eclipse Marketplace: WatchDog Plugin. `https://marketplace.eclipse.org/content/testroots-watchdog`. Accessed June 6, 2017.

[207] IntelliJ Marketplace: WatchDog Plugin. `https://plugins.jetbrains.com/plugin/7828-watchdog`. Accessed June 6, 2017.

[208] ReSharper Plugin Gallery: FeedBaG++ Plugin. `https://resharper-plugins.jetbrains.com/packages/KaVE.Project/`. Accessed June 6, 2017.

[209] Code Trails Marketplace: WatchDog Plugin. `http://www.codetrails.com/blog/test-analytics-testroots-watchdog`. Accessed June 6, 2017.

[210] KAVE Datasets: Interaction Data, March 1, 2017. `http://www.kave.cc/datasets/`. Accessed June 6, 2017.

[211] G. Rothermel and S. Elbaum. Putting your best tests forward. *IEEE Software*, 20(5):74–77, Sept 2003.

[212] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Datattreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of the 10th joint meeting on the Foundations of Software Engineering*, pages 770–781. ACM, 2015.

[213] Fabio Palomba and Andy Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE, 2017.

[214] Torleif Halkjelsvik and Magne Jørgensen. From origami to software development: A review of studies on judgment-based predictions of performance time. *Psychological Bulletin*, 138(2):238, 2012.

[215] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of ci build failures: an open source and a financial organization perspective. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages pages 183–193. IEEE.

[216] Andrew Patterson, Michael Kölling, and John Rosenberg. Introducing unit testing with BlueJ. *ACM SIGCSE Bulletin*, 35(3):11–15, June 2003.

[217] Esther Derby, Diana Larsen, and Ken Schwaber. *Agile retrospectives: Making good teams great.* Pragmatic Bookshelf, 2006.

[218] Victor Hurdugaci and Andy Zaidman. Aiding software developers to maintain developer tests. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 11–20. IEEE, 2012.

[219] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *Proceedings International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 195–204. IEEE, 2014.

[220] Simone Romano, Davide Fucci, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo. Findings from a multi-method study on test-driven development. *Information and Software Technology*, 2017.

[221] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. An empirical evaluation and comparison of manual and automated test selection. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 361–372. ACM, 2014.

[222] Abraham Bookstein. Informetric distributions, part i: Unified overview. *Journal of the American Society for Information Science (1986-1998)*, 41(5):368, 1990.

[223] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[224] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 483–493. IEEE Press, 2015.

[225] Jeff Anderson, Saeed Salem, and Hyunsook Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 142–151, New York, NY, USA, 2014. ACM.

[226] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 102–111. ACM, 2014.

[227] Inozemtseva, Laura M. M. *Data Science for Software Maintenance*. PhD thesis, 2017.

[228] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, (9):43–50, 2005.

[229] Neil Borle, Meysam Feghhi, and Abram Hindle. Analysis of test driven development on sentiment and coding activities in github repositories. *PeerJ PrePrints*, 4:e1920, 2016.

[230] Simone Romano, Davide Fucci, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo. Results from an ethnographically-informed study in the context of test driven development. In *Proceedings of the 20th International Conference on Evaluation and*

Assessment in Software Engineering, EASE '16, pages 10:1–10:10, New York, NY, USA, 2016. ACM.

[231] Jerod W Wilkerson, Jay F Nunamaker Jr, and Rick Mercer. Comparing the defect reduction benefits of code inspection and test-driven development. *Software Engineering, IEEE Transactions on*, 38(3):547–560, 2012.

[232] Andy Oram and Greg Wilson. *Making software: What really works, and why we believe it.* " O'Reilly Media, Inc.", 2010.

[233] Sami Kollanus. Test-driven development-still a promising approach? In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 403–408. IEEE, 2010.

[234] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering*, 31(3):226–237, 2005.

[235] Laurie Williams, E Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 34–45. IEEE, 2003.

[236] Nachiappan Nagappan, E Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3):289–302, 2008.

[237] Matthias M Müller and Frank Padberg. About the return on investment of test-driven development. In *5th International Workshop on Economic-driven Software Engineering Research*, page 26, 2003.

[238] Gerardo Canfora, Aniello Cimitile, Felix Garcia, Mario Piattini, and Corrado Aaron Visaggio. Evaluating advantages of test driven development: a controlled experiment with professionals. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 364–371. ACM, 2006.

[239] Joint Task Force on Computing Curricula, IEEE Computer Society, and Association for Computing Machinery. Curriculum guidelines for undergraduate degree programs in software engineering. `http://www.acm.org/binaries/content/assets/education/se2014.pdf`. Accessed June 6, 2017.

[240] David S Janzen and Hossein Saiedian. Does test-driven development really improve software design quality? *Software, IEEE*, 25(2):77–84, 2008.

[241] David Heinemeier Hansson. TDD is dead. long live testing. `http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html`. Accessed June 6, 2017.

[242] David Heinemeier Hansson. Test-induced design damage. `http://david.heinemeierhansson.com/2014/test-induced-design-damage.html`. Accessed June 6, 2017.

[243] David Heinemeier Hansson, Kent Beck, and Martin Fowler. Is TDD dead? `https://youtu.be/z9quxZsLcfo`. Accessed April 13, 2016.

[244] Android Studio Documentation: Test Your App. `https://developer.android.com/studio/test/index.html`. Accessed June 12, 2017.

[245] Terry Connolly and Doug Dean. Decomposed versus holistic estimates of effort required for software writing tasks. *Management Science*, 43(7):1029–1045, 1997.

[246] Ilinca Ciupa. Test studio: An environment for automatic test generation based on design by contract, 2004.

[247] André N. Meyer, Thomas Fritz, Gail C. Murphy, and Thomas Zimmermann. Software developers' perceptions of productivity. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 19–29. ACM, 2014.

[248] Les Hatton. How accurately do engineers predict software maintenance tasks? *Computer*, (2):64–69, 2007.

[249] Michael M Roy, Nicholas JS Christenfeld, and Craig RM McKenzie. Underestimating the duration of future events: memory incorrectly used or memory bias? *Psychological bulletin*, 131(5):738, 2005.

[250] Michael M Roy, Scott T Mitten, and Nicholas JS Christenfeld. Correcting memory improves accuracy of predicted task duration. *Journal of Experimental Psychology: Applied*, 14(3):266, 2008.

[251] Magne Jørgensen and Dag Sjøberg. Generalization and theory-building in software engineering research. *Empirical Assessment in Software Eng. Proc*, pages 29–36, 2004.

[252] Dag IK Sjoberg, Tore Dyba, and Magne Jorgensen. The future of empirical methods in software engineering research. In *Future of Software Engineering, 2007. FOSE'07*, pages 358–378. IEEE, 2007.

[253] André N Meyer, Thomas Fritz, Gail C Murphy, and Thomas Zimmermann. Software developers' perceptions of productivity. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 19–29. ACM, 2014.

[254] Emily IM Collins, Anna L Cox, Jon Bird, and Daniel Harrison. Social networking use and rescuetime: the issue of engagement. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pages 687–690. ACM, 2014.

[255] Philip M Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen, and William EJ Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 25th international Conference on Software Engineering*, pages 641–646. IEEE Computer Society, 2003.

[256] Lile Hattori and Michele Lanza. Syde: a tool for collaborative software development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 235–238. ACM, 2010.

[257] Romain Robbes and Michele Lanza. Spyware: a change-aware development toolset. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 847–850. ACM, 2008.

[258] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, pages 552–576, 2013.

[259] Roberto Minelli, Andrea Mocci, Michele Lanza, and Lorenzo Baracchi. Visualizing developer interactions. In *Proceedings of the Working Conference on Software Visualization (VISSOFT)*, pages 147–156. IEEE, 2014.

[260] Hongbing Kou, Philip M Johnson, and Hakan Erdogmus. Operational definition and automated inference of test-driven development with zorro. *Automated Software Engineering*, 17(1):57–85, 2010.

[261] Philip M Johnson. Searching under the streetlight for useful software analytics. *IEEE software*, (4):57–63, 2013.

[262] Oren Mishali, Yael Dubinsky, and Shmuel Katz. The TDD-Guide training and guidance tool for test-driven development. In *Agile Processes in Software Engineering and Extreme Programming*, pages 63–72. Springer, 2008.

[263] Yihong Wang and Hakan Erdogmus. The role of process measurement in test-driven development. In *4th Conference on Extreme Programming and Agile Methods*, 2004.

[264] P.S. Kochhar, T.F. Bissyande, D. Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 103–112. IEEE, 2013.

[265] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 492–501. ACM, 2006.

[266] Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. Enablers, inhibitors, and perceptions of testing in novice software teams. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 30–40. ACM, 2014.

[267] Paul Dan Marinescu, Petr Hosek, and Cristian Cadar. Covrig: a framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 93–104. ACM, 2014.

[268] Robert Feldt. Do system test cases grow old? In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 343–352. IEEE, 2014.

[269] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Trans. Software Eng.*, 40(11):1100–1125, 2014.

[270] Michaela Greiler, Arie van Deursen, and M Storey. Test confessions: a study of testing practices for plug-in systems. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 244–254. IEEE, 2012.

[271] Martin Fowler and Matthew Foemmel. Continuous integration. 2006. `http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf`.

[272] Michael A Cusumano and Richard W Selby. Microsoft secrets: how the world's most powerful software company creates technology, shapes markets, and manages people, 1997.

[273] Kent Beck. *Extreme programming explained: embrace change.* Addison-Wesley Professional, 2000.

[274] Martin Brandtner, Emanuel Giger, and Harald C. Gall. Sqa-mashup: A mashup framework for continuous integration. *Information & Software Technology*, 65:97–113, 2015.

[275] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. IEEE, 2013.

[276] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 598–608. IEEE, 2015.

[277] Jonathan Rasmusson. Long build trouble shooting guide. In Carmen Zannier, Hakan Erdogmus, and Lowell Lindstrom, editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2004*, volume 3134 of *LNCS*, pages 13–21. Springer, 2004.

[278] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at Google). In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 724–734. ACM, 2014.

[279] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 401–405. IEEE, 2014.

[280] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 805–816. ACM, 2015.

[281] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007.

[282] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[283] Per Runeson, Carina Andersson, Thomas Thelin, Anneliese Andrews, and Tomas Berling. What do we know about defect detection methods? *Software, IEEE*, 23(3):82–90, 2006.

[284] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *IEEE Computer*, 38(9):43–50, 2005.

[285] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.

[286] R. Ablett, E. Sharlin, F. Maurer, J. Denzinger, and C. Schock. Buildbot: Robotic monitoring of agile software development teams. In *Proceedings of the International Symposium on Robot and Human interactive Communication (RO-MAN)*, pages 931–936. IEEE, 2007.

[287] R.O. Rogers. Scaling continuous integration. In *Extreme programming and agile processes in software engineering*, number 3092 in LNCS, pages 68–76. 2004.

[288] Christina Watters and Peter Johnson. Version numbering in single development and test environment, December 29 2011. US Patent App. 13/339,906.

[289] What is Travis CI. `https://github.com/travis-ci/travis-ci/blob/2ea7620f4be51a345632e355260b22511198ea64/README.textile#goals`.

[290] Travis CI. Travis ci api documentation. `http://docs.travis-ci.com/user/getting-started/`.

[291] Travis CI. Travis ci plans and costs. `https://travis-ci.com/plans`.

[292] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[293] Travis CI. Travis ci rest api. `https://api.travis-ci.org/`.

[294] Moritz Beller (Inventitech). Travis ci memory leak issue. `https://github.com/travis-ci/travis.rb/issues/310`.

[295] Travis CI. Undocumented travis log archive server. `http://s3.amazonaws.com/archive.travis-ci.org`.

[296] O. Tange. GNU Parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[297] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Engineering*, 20(6):1587–1633, 2015.

[298] David Chelimsky, Dave Astels, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends*. Pragmatic Bookshelf, 1st edition, 2010.

[299] András Vargha and Harold D Delaney. A critique and improvement of the CL common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[300] Vigdis By Kampenes, Tore Dybå, Jo E Hannay, and Dag IK Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11):1073–1086, 2007.

[301] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github (extended version). *Empirical Software Engineering*, 2015. Accepted for publication.

[302] Github. Language trends on GitHub. `https://github.com/blog/2047-language-trends-on-github`.

[303] Scala. Github repository of scala. `https://github.com/scala/scala`.

[304] Scala. The ci server of scala. `https://scala-ci.typesafe.com/`.

[305] Mathias Meyer. The Travis CI blog: Supporting the ruby ecosystem, together. `https://blog.travis-ci.com/2016-02-03-supporting-the-ruby-ecosystem-together`.

[306] L. Tratt and R. Wuyts. Guest editors' introduction: Dynamically typed languages. *IEEE Software*, 24(5):28–30, 2007.

[307] Diomidis Spinellis. *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Addison-Wesley, 2016.

[308] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *Proceedings of the Joint European Conferences on Theory and Practice of Software and 17th international conference on Compiler construction (CC'08/ETAPS'08)*, pages 147–162. Springer, 2008.

[309] Brian W Kernighan and Phillip James Plauger. The elements of programming style. *The elements of programming style, by Kernighan, Brian W.; Plauger, PJ New York: McGraw-Hill, c1978.*, 1978.

[310] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[311] Benjamin Siegmund, Michael Perscheid, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 269–274. IEEE, 2014.

[312] P.W. Oman, C.R. Cook, and M. Nanja. Effects of programming experience in debugging semantic errors. *Journal of Systems and Software*, 9(3):197––207, 1989.

[313] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, August 2011.

[314] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 121–130. IEEE, 2011.

[315] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 61–72. ACM, 2010.

[316] Muhammad Zubair Malik, Junaid Haroon Siddiqi, and Sarfraz Khurshid. Constraint-based program debugging using data structure repair. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 190–199. IEEE, 2011.

[317] Henry Lieberman. The debugging scandal and what to do about it (introduction to the special section). *Commun. ACM*, 40(4):26–29, 1997.

[318] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*. Free Software Foundation, 10 edition, 2011.

[319] Ed Burnette. *Eclipse IDE Pocket Guide*. O'Reilly Media, Inc., 2005.

[320] Even Adams and Steven S. Muchnick. dbxtool: A window-based symbolic debugger for Sun workstations. *Software: Practice and Experience*, 16(7):653–669, July 1986.

[321] Norman Matloff and Peter Jay Salzman. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, San Francisco, 2008.

[322] Bert Beander. VAX DEBUG: An interactive, symbolic, multilingual debugger. In M.S. Johnson, editor, *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 173–179. ACM SIGSOFT/SIGPLAN, March 1983.

[323] Bill Tuthill and Kevin J. Dunlap. Debugging with dbx. In *UNIX Programmer's Supplementary Documents, Volume 1*. Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.

[324] Andreas Zeller and Dorothea Lütkehaus. DDD – a free graphical front-end for unix debuggers. *ACM Sigplan Notices*, 31(1):22–27, 1996.

[325] Andrew Ko and Brad Myers. Debugging reinvented. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 301–310. IEEE, 2008.

[326] David J Gilmore. Models of debugging. *Acta psychologica*, 78(1):151–172, 1991.

[327] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[328] Frank Eichinger, Klaus Krogmann, Roland Klug, and Klemens Böhm. Software-defect localisation by mining dataflow-enabled call graphs. In *Machine Learning and Knowledge Discovery in Databases*, pages 425–441. Springer, 2010.

[329] Saeed Parsa, Mojtaba Vahidi-Asl, Somaye Arabi, and Behrouz Minaei-Bidgoli. Software fault localization using elastic net: A new statistical approach. In *Advances in Software Engineering*, pages 127–134. Springer, 2009.

[330] David Abramson, Clement Chu, Donny Kurniawan, and Aaron Searle. Relative debugging in an integrated development environment. *Software: Practice and Experience*, 39(14):1157–1183, 2009.

[331] Cheng Zhang, Juyuan Yang, Dacong Yan, Shengqian Yang, and Yuting Chen. Automated breakpoint generation for debugging. *Journal of Software*, 8(3), 2013.

[332] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 309–319. ACM, 2012.

[333] Yan Lei, Xiaoguang Mao, Ziying Dai, and Chengsong Wang. Effective statistical fault localization using program slices. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 1–10, July 2012.

[334] Saeed Parsa, Farzaneh Zareie, and Mojtaba Vahidi-Asl. Fuzzy clustering the backward dynamic slices of programs to identify the origins of failure. In *Experimental Algorithms*, pages 352–363. Springer, 2011.

[335] Michael Perscheid and Robert Hirschfeld. Follow the path: Debugging tools for test-driven fault navigation. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 446–449. IEEE, 2014.

[336] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Practical isolation of failure-inducing changes for debugging regression faults. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 20–29. IEEE, 2012.

[337] Alessandro Orso. Automated debugging: Are we there yet? `https://www.youtube.com/watch?v=WJHQnzLpVXk&feature=youtu.be`, 2014. Accessed July 11, 2016].

[338] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1):83–110, 2016.

[339] Gail C Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Elipse IDE? *IEEE software*, 23(4):76–83, 2006.

[340] Chris Parnin and Spencer Rugaber. Resumption strategies for interrupted programming tasks. *Software Quality Journal*, 19(1):5–34, 2011.

[341] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 383–392. IEEE, 2013.

[342] David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, Austin Z. Henley, Jamie Macbeth, Charles Hill, and Amber Horvath. To fix or to learn? how production bias affects developers' information foraging during debugging. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 11–20. IEEE, 2015.

[343] David J Piorkowski, Scott D Fleming, Irwin Kwan, Margaret M Burnett, Christopher Scaffidi, Rachel KE Bellamy, and Joshua Jordahl. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3063–3072. ACM, 2013.

[344] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. How developers debug software the dbgbench dataset: Poster. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE Companion)*, pages 244–246. IEEE, 2017.

[345] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and Symopsium on the Foundations of Software Engineering (ESEC/FSE)*. IEEE, 2017.

[346] Fabio Petrillo, Zéphyrin Soh, Foutse Khomh, Marcelo Pimenta, Carla Freitas, and Yann-Gaël Guéhéneuc. Understanding interactive debugging with swarm debug infrastructure. In *Proceedings of the 24th International Conference on Program Comprehension*, pages 1–4. ACM, 2016.

[347] Brian W Kernighan. *UNIX for Beginners.* Bell Laboratories Murray Hill, NJ, 1978.

[348] Diomidis Spinellis. Debuggers and logging frameworks. *IEEE Software*, 23(3):98–99, May/June 2006.

[349] Marcel Das, Peter Ester, and Lars Kaczmirek. *Social and behavioral research and the internet: Advances in applied methods and research strategies.* Routledge, 2010.

[350] Donna Spencer. *Card sorting: Designing usable categories.* Rosenfeld Media, 2009.

[351] Priscilla E Greenwood and Michael S Nikulin. *A guide to chi-squared testing*, volume 280. John Wiley & Sons, 1996.

[352] Cay S Horstmann and Gary Cornell. *Core Java 2: Volume I, Fundamentals.* Pearson Education, 2002.

[353] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 485–495. IEEE Press, 2012.

[354] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The moldable debugger: A framework for developing domain-specific debuggers. In *International Conference on Software Language Engineering*, pages 102–121. Springer, 2014.

[355] Daniel A. Keim. Visual exploration of large data sets. *Commun. ACM*, 44(8):38–44, 2001.

[356] Organisation for Economic Co-Operation and Development. Average annual hours actually worked per worker. `http://stats.oecd.org/index.aspx?DataSetCode=ANHRS`, 2015. Accessed July 11, 2016].

[357] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC counting standard. In *COCOMO II Forum*, volume 2007, 2007.

[358] Devon H. O'Dell. The debugging mind-set. *Communications of the ACM*, 60(6):40–45, 2017.

[359] Boris Beizer. Software testing techniques. *New York, ISBN: 0-442-20672-0*, 1990.

[360] Remodularizing Java programs for improved locality of feature implementations in source code. *Science of Computer Programming*, 77(3):131–151, 2012.

[361] Guillaume Pothier and Éric Tanter. Back to the future: Omniscient debugging. *IEEE software*, 26(6):78–85, 2009.

[362] Chronon Systems. Chronon, a DVR for Java. `http://chrononsystems.com/`, 2017. Accessed November 9, 2017.

[363] Apple Inc. Xcode 8. `https://developer.apple.com/xcode/`, 2015. Accessed August 24, 2016.

[364] Gina Masullo Chen. Tweet this: A uses and gratifications perspective on how active twitter use gratifies a need to connect with others. *Computers in Human Behavior*, 27(2):755–762, 2011.

[365] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35. IEEE Press, 2015.

[366] David AW Soergel. Rampant software errors may undermine scientific results. *F1000Research*, 3, 2014.

[367] Erik Meijer. Goto 2015 • one hacker way.

[368] Miju Han. Introducing security alerts on github, 2017. `https://github.com/blog/2470-introducing-security-alerts-on-github`. Accessed January 26, 2018.

[369] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald Gall. Context is king: The developer perspective on the usage of static analysis tools. 2018.

[370] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 334–344. IEEE Press, 2017.

[371] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 345–355. IEEE Press, 2017.

[372] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 821–830, New York, NY, USA, 2017. ACM.

[373] Ansong Ni and Ming Li. Cost-effective build outcome prediction using cascaded classifiers. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 455–458. IEEE Press, 2017.

[374] Ekaba Bisong, Eric Tran, and Olga Baysal. Built to last or built too fast?: evaluating prediction models for build times. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 487–490. IEEE Press, 2017.

[375] Marcel Rebouças, Renato O Santos, Gustavo Pinto, and Fernando Castor. How does contributors' involvement influence the build status of an open-source software project? In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 475–478. IEEE Press, 2017.

[376] Rodrigo Souza and Bruno Silva. Sentiment analysis of travis ci builds. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 459–462. IEEE Press, 2017.

[377] Md Rakibul Islam and Minhaz F Zibran. Insights into continuous integration build failures. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 467–470. IEEE Press, 2017.

[378] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 495–498. IEEE Press, 2017.

[379] Abigail Atchison, Christina Berardi, Natalie Best, Elizabeth Stevens, and Erik Linstead. A time series analysis of travistorrent builds: to everything there is a season. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 463–466. IEEE Press, 2017.

[380] Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. What are the testing habits of developers? a case study in a large it company. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 58–68. IEEE, 2017.

[381] Dag IK Sjøberg, Tore Dybå, Bente CD Anda, and Jo E Hannay. Building theories in software engineering. In *Guide to advanced empirical software engineering*, pages 312–336. Springer, 2008.

Glossary

ASAT Automated Static Analysis Tools (described in Chapter 2).

CI Continuous Integration.

FDD Feedback-Driven Development, a model of the modern code creation cycle that involves acquiring and integrating feedback from multiple sources and passing quality gates in a highly customizable way (described in Chapter 1).

GDC General Defect Classification (described in Chapter 2).

GHTorrent A scalable, queriable, offline mirror of data offered through the GitHub REST API.

GitHub The most widely used collaborative software building platform (2018), free to use for OSS.

IDE Integrated Development Environment. Common examples include Eclipse, IntelliJ, Visual Studio, or Visual Code.

OSS Open-Source Software.

SLOC Source Lines of Code, a measure for the size of a software system, without whitespaces and comments.

TDD Test-Driven Development, the method of continuously co-evolving test and production code, originally proposed by Beck [5].

TestRoots Name of the NWO project under which this thesis has been carried out.

TGD Test-Guided Development, the act of loosely guiding one's development efforts with the help of tests (described in Chapter 4).

Travis CI The most wide-spread CI server on GitHub (2018), free to use for OSS.

TravisTorrent A free and open database of analyzed CI build logs from Travis CI (described in Chapter 5).

UAV Unified ASAT Visualizer (described in Chapter 2).

WatchDog Infrastructure and series of plugins to assess telemetry data from developers' behavior in the IDE (described in Chapters 4 and 6).

Curriculum Vitæ

Moritz Marc Beller

1988/07/30	Date of birth in Schweinfurt, Germany

Education

1/2014–1/2018	Ph.D. Student, Software Engineering Research Group, Delft University of Technology, The Netherlands, *An Empirical Analysis of Feedback-Driven Development* Supervisor: Dr. Georgios Gousios Promotors: Prof. Dr. Arie van Deursen, Dr. Andy Zaidman
10/2011–10/2013	M.Sc. Computer Science, Technische Universität München, Germany, Thesis: *Quantifying Continuous Code Reviews*, 1.0
9/2008–9/2011	B.Sc. Computer Science, Technische Universität München, Germany, Thesis: *Static Validation of ConQAT Architecture Descriptions*, 1.0
8/2010–3/2011	Erasmus Abroad Studies, Linköpings Universitet, Sweden
9/1999–6/2008	University Entrance Diploma, Celtis-Gymnasium, Schweinfurt, Germany, 1.7

Experience

6/2017–9/2017	Research Intern, Empirical Software Engineering Group, Microsoft Research, Redmond (WA), USA, Supervisor: Dr. Thomas Zimmermann
1/2016–2/2016	Visiting Researcher, The Computer Human Interaction & Software Engineering Lab of Prof. Dr. Margaret-Anne Storey,

	University of Victoria, Canada
10/2011–3/2013	Student Intern as Software Engineer and Consultant on ConQAT and Teamscale (`teamscale.eu`), CQSE GmbH, München, Germany
1/2011–3/2011	Student Intern as Software Engineer in "Research & Development," Sick IVP A/B, Linköping, Sweden
8/2009–9/2009	Trainee, SKF Sverige A/B, SKF Competence Centre, Gothenborg, Sweden
2/2009–9/2009	Student Researcher, Chair for Software and Systems Engineering, Technische Universität München, Germany
7/2007–8/2007	Student Intern, Central Department Research & Development, SICK AG, Waldkirch, Germany

Academic Service

Chair	Mining Challenge Chair, MSR 2017
	Social Media Co-Chair, SCAM 2016
PC Member	ISSTA 2018, Artifact Evaluation MSR 2018, Mining Challenge MSR 2018 ESEC/FSE, Tool Track, 2017 SANER, Tool Track, 2017 SCAM, Tool Track, 2015
Reviewer	JSS 2016, 2017 EMSE 2015, 2017 ESEC/FSE 2015, 2017 SANER 2015, 2016 MSR 2014, 2016 SCAM 2014
Council Member	PhD council of the Institute for Programming Research and Algorithmics (IPA)
(Co-)Supervisor	Radjino Bholanath's Master Thesis "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," 2014–2015

	Niels Spruit's Master Thesis "What Programmers Know About Debugging And How They Use Their IDE Debuggers," 2016
	Igor Levaja's Master Thesis on Developer Testing, 2015–2016
	Context Project of a group of five Bachelor students, who created a static analysis visualization tool, 2016
	Bachelor End Project of a group of three students developing an app for flight risk assessment, 2016
Teaching Assistant	Software Testing and Quality Engineering, Prof. Dr. Arie van Deursen, 2014 Software Engineering Methods, Dr. Alberto Bacchelli, 2014

Invited Talks & Lectures

Guest Lectures	*When, How, and Why Developers (Do Not) Test* in Dr. Ali Mesbah's Software Verification and Testing course at the University of British Columbia (UBC), Vancouver, Canada, 22.2.2016
	Empirical Studies in Software Engineering: 3 Examples in Dr. Alberto Bacchelli's Mining Software Repositories course, Delft, The Netherlands, 7.12.2015
Industry Talks	*Manual And Automated Static Analysis in Open Source Software: Prevalence and Usage* at Microsoft Research, Redmond, USA, 8.2.2016
	How (Much) Do Developers Test? in Software Industry Conference (SINC), Bussum, The Netherlands, 12.3.2015.
	TestRoots: Learn From Past Test Failures at TNG Techday, Munich, Germany, 28.11.2014.
	When, How, and Why Developers (Do Not) Test at the Project Quality Day, EclipseCon Europe 2015, Ludwigsburg, Germany, 4.11.2015.
	How (Much) Do Developers Test? at Eclipse Democamp, Kassel, Germany, 9.12.2014.
	Test Analytics: How Much Testing Is Enough? in BOF-Session, EclipseCon Europe 2014, Ludwigsburg, Germany, 28.10.2014.

List of Publications

16. *Moritz Beller*: Toward an Empirical Theory of Feedback-Driven Development. To appear in 40th International Conference on Software Engineering (ICSE), Student Research Competition (SRC), Gothenborg, Sweden, 2018. Acceptance Rate 43% (10/23)

15. *Moritz Beller,* Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the Dichotomy of Debugging Behavior Among Programmers. To appear in 40th International Conference on Software Engineering (ICSE), Gothenborg, Sweden, 2018. Acceptance Rate 21% (105/502)

14. *Moritz Beller,* Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, Andy Zaidman: Developer Testing in the IDE: Patterns, Beliefs, and Behavior. To appear in IEEE Transactions on Software Engineering (TSE), 2018.

13. Alberto Bacchelli, *Moritz Beller*: Double-Blind Review in Software Engineering Venues. In 39th International Conference on Software Engineering (ICSE), Introspection track, Buenos Aires (Argentina), 2017.

12. *Moritz Beller,* Georgios Gousios, Andy Zaidman: Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In 14th International Conference on Mining Software Repositories (MSR), Buenos Aires (Argentina), 2017. Acceptance Rate 31% (37/121)

11. *Moritz Beller,* Georgios Gousios, Andy Zaidman: TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In 14th International Conference on Mining Software Repositories (MSR), Buenos Aires (Argentina), 2017. Acceptance Rate 25% (1/4)

10. Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, *Moritz Beller,* Andy Zaidman: UAV: Warnings from Multiple Automated Static Tools at a Glance. In 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), Tool track, Klagenfurt (Austria), 2017. Acceptance Rate 55% (12/22)

9. *Moritz Beller,* Andy Zaidman, Andrey Karpov, Rolf A. Zwaan: The Last Line Effect Explained. In Empirical Software Engineering (EMSE), 2016.

8. Carmine Vassallo, Fiorella Zampetti, Daniele Romano, *Moritz Beller,* Annibale Panichella, Massimiliano Di Penta, Andy Zaidman: Continuous Delivery Practices in a Large Financial Organization. In 32nd International Conference on Software Maintenance and Evolution (ICSME), Industrial track, Raleigh (USA), 2016. Acceptance Rate 55% (12/22)

7. Sebastiano Panichella, Annibale Panichella, *Moritz Beller,* Andy Zaidman, Harald Gall: The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation. In 38th International Conference on Software Engineering (ICSE), Austin (USA), 2016. Acceptance Rate 19% (101/530)

6. *Moritz Beller,* Igor Levaja, Annibale Panichella, Georgios Gousios, Andy Zaidman: How to Catch 'Em All: WatchDog, a Family of IDE Plug-Ins to Assess Testing. In 3rd International Workshop on Software Engineering Research and Industrial Practice (SER&IP 2016), Austin (USA), 2016. Acceptance Rate 32% (10/31)

5. *Moritz Beller,* Radjino Bholanath, Shane McIntosh, Andy Zaidman: Analyzing the State of Static Analysis: A Large-Scale Evaluation. In Open Source Software in 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka (Japan), 2016. Acceptance Rate 37% (52/140)

4. *Moritz Beller,* Georgios Gousios, Annibale Panichella, Andy Zaidman: When, How, and Why Developers (Do Not) Test in Their IDEs. In 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Bergamo (Italy), 2015. Acceptance Rate 25% (74/291)

3. *Moritz Beller,* Andy Zaidman, Andrey Karpov: The Last Line Effect. In 23rd International Conference on Program Comprehension (ICPC), Early Research Achievements (ERA) track, Florence (Italy), 2015. Acceptance Rate 32% (7/22)

2. *Moritz Beller,* Georgios Gousios, Andy Zaidman: How (Much) Do Developers Test? In 37th International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER) track, Florence (Italy), 2015. Acceptance Rate 18% (25/135)

1. *Moritz Beller,* Alberto Bacchelli, Andy Zaidman, Elmar Jürgens: Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix? In 11th Working Conference on Mining Software Repositories (MSR), Hyderabad (India), 2014. Acceptance Rate 34% (29/85)

Included in this thesis.
Won a best paper, tool demonstration, or proposal award.

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*.

Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network.* Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16