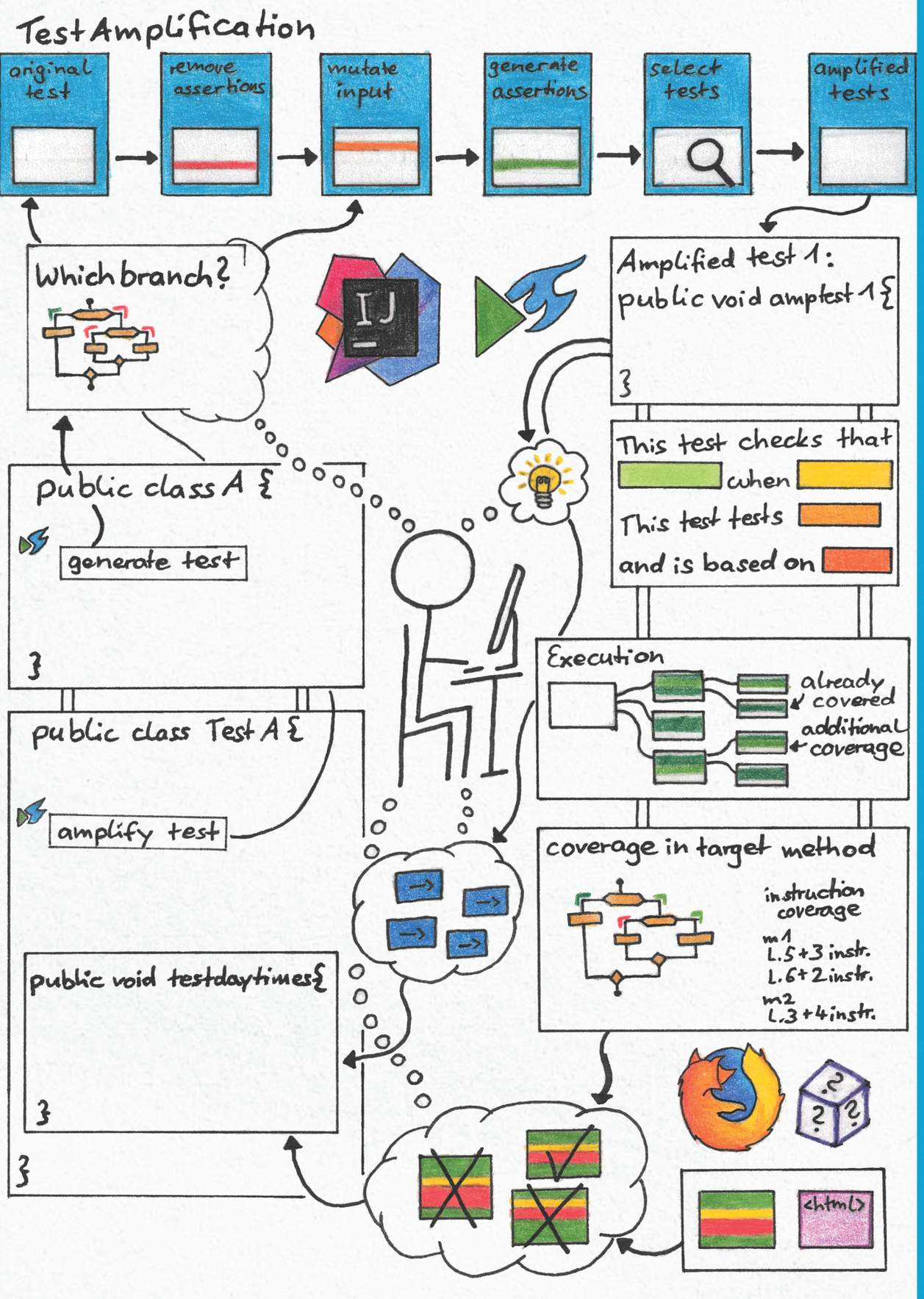
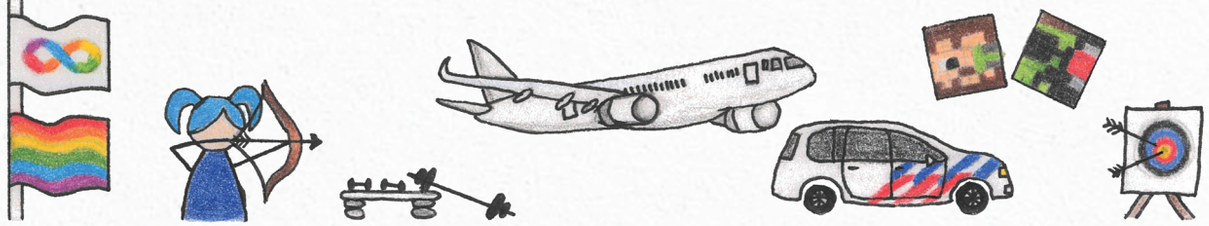


Carolin Brandt

Test Amplification For and With Developers



Test Amplification For and With Developers

Carolin Brandt

Test Amplification For and With Developers

Test Amplification For and With Developers

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 maandag 8 juli 2024 om 17:30 uur

door

Carolin Elisabeth BRANDT

Master of Science in Software Engineering,
Universität Augsburg, Technische Universität München, en
Ludwig-Maximilians-Universität München, Duitsland,
geboren te München, Duitsland.

Dit proefschrift is goedgekeurd door de

promotoren: Prof. dr. A.E. Zaidman, Prof. dr. A. van Deursen

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. A.E. Zaidman,	Technische Universiteit Delft, promotor
Prof. dr. A. van Deursen,	Technische Universiteit Delft, promotor
Onafhankelijke leden:	
Prof. dr. B. Baudry,	Universtité de Montréal
Prof. dr. A. Zeller,	CISPA Helmholtz Center for Information Security and Saarland University
Prof. dr. X. Devroey,	University of Namur
Prof. dr. K. Langendoen,	Technische Universiteit Delft
Dr. B. Külahçıoğlu Özkan,	Technische Universiteit Delft
Prof. dr. ir. Fernando Kuipers,	Technische Universiteit Delft, reservelid

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Keywords: Test Amplification, Developer-Centric Design, Software Testing

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

Cover: Carolin Brandt

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

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

ISBN 978-94-6366-886-6

An electronic version of this dissertation is available at
`https://doi.org/10.4233/uuid:aedfd7b6-f9ae-4b76-9122-29e43995d36f`.

Contents

Summary **ix**

Samenvatting **xi**

Acknowledgments **xiii**

1 Introduction **1**

2 Developer-Centric Test Amplification: The Interplay Between Automatic Generation and Human Exploration **13**

2.1 Creating Developer-Centric Test Amplification. 17
2.2 Bringing Test Amplification to the Developer (IDE) 18
2.2.1 Developer-Centric Amplification With DSpot 20
2.2.2 Test Exploration Plugin *TestCube* 23
2.3 Study Design 25
2.3.1 Preparation 26
2.3.2 Interview Procedure 27
2.3.3 Data Collection and Analysis. 27
2.4 Results 28
2.4.1 Participants 28
2.4.2 RQ1: What Are the Key Factors to Make Amplified Test Cases Suited for Developer-Centric Test Amplification? 28
2.4.3 RQ2: What Are the Key Factors to Make Test Exploration Tools Suited for Developer-Centric Test Amplification? 32
2.4.4 RQ3: What Information Do Developers Seek While Exploring Amplified Test Cases? 35
2.4.5 RQ4: What Value Does Developer-Centric Test Amplification Bring to Developers? 37
2.5 Discussion and Recommendations 38
2.6 Threats to Validity 40
2.7 Related Work. 41
2.7.1 Understandability of Test Cases 41
2.7.2 Test Generation Tools Integrated in the IDE 43
2.7.3 Interactive Test Generation. 43
2.8 Conclusion and Future Work 44

3 How Does This New Developer Test Fit In? A Visualization to Understand Amplified Test Cases **45**

3.1 Developer-Centric Test Amplification 48
3.2 The Test Impact Graph 48
3.2.1 Method Nodes 48

3.2.2 Call Edges . . . 49
3.2.3 Default Layout . . . 49
3.2.4 Design Rationale . . . 50
3.2.5 Implementation . . . 50
3.3 Think-Aloud Study . . . 51
3.3.1 Study Design . . . 51
3.3.2 Study Execution . . . 52
3.3.3 General Observations . . . 53
3.4 RQ1: Which Information Do Developers Seek From the TestImpactGraph? 53
3.4.1 What Does It Do? Understanding the Test Case . . . 53
3.4.2 Should I Test This? Provide Scope of Where Code Is From . . . 54
3.4.3 Who Tests This Already? Support Exploring Other Tests . . . 55
3.5 RQ2: Which Features of the TestImpactGraph Help Developers Access This Information? . . . 55
3.5.1 Code Nodes: As Close to the IDE as Possible . . . 55
3.5.2 Default View: Provide Confidence to See Everything Relevant . . . 56
3.5.3 Where Was I? Providing and Keeping Context . . . 56
3.5.4 Where Does This Connect? Clarifying Edges . . . 57
3.6 RQ3: What Observations Related to Test Coverage Arise When Inspecting a Developer Test Through the TestImpactGraph? . . . 57
3.6.1 Should This Not Already Be Unit-Tested? Deep and Accidental Coverage . . . 57
3.6.2 What Is Executed Here? Instruction Coverage Visualized Per Line . 58
3.7 Discussion . . . 59
3.7.1 Test Review . . . 59
3.7.2 Differential Code Coverage . . . 59
3.7.3 Refined Coverage Insights . . . 60
3.7.4 Relevance of Deep Coverage for Test Descriptions . . . 60
3.7.5 Threats to Validity . . . 60
3.8 Related Work: Test Visualizations . . . 61
3.9 Conclusion and Future Work . . . 63

4 Shaken, Not Stirred. How Developers Like Their Amplified Tests 65
4.1 Developer-Centric Test Amplification . . . 67
4.2 Automatic Post-Processing for developer-centric test amplification. . . . 69
4.2.1 Prettifier module . . . 70
4.2.2 Descriptions for Amplified Tests . . . 71
4.3 Open Source Contribution Study . . . 72
4.3.1 Repository Selection . . . 73
4.3.2 Running the Test Amplification . . . 73
4.3.3 Manual Selection and Editing . . . 73
4.3.4 Contributing Back the Tests . . . 74
4.3.5 Data Analysis . . . 74

4.4 Results . . . 75
4.4.1 Running the Test Amplification . . . 75
4.4.2 **RQ1.1:** On which criteria do we select a candidate test to include in the test suite? . . . 75
4.4.3 **RQ1.2:** Which manual edits do we perform to improve the tests before submission? . . . 78
4.4.4 **RQ2.1:** Which changes are proposed during the pull request discussion? . . . 80
4.4.5 **RQ2.2:** What kind of information is requested by the maintainers during the pull request discussion? . . . 81
4.4.6 **RQ2.3:** How do the maintainers justify their judgment over the amplified tests during the pull request discussion? . . . 82
4.5 Discussion . . . 83
4.5.1 Guidelines for Developers to Select and Edit Amplified Tests . . . 83
4.5.2 Relation to Existing Literature . . . 85
4.5.3 Implications for Practitioners . . . 87
4.5.4 Implications for Researchers and Tool Designers . . . 88
4.5.5 Threats to Validity . . . 89
4.6 Related Work . . . 90
4.7 Conclusion and Future Work . . . 91

5 When to Let the Developer Guide: Trade-offs Between Open and Guided Test Amplification 93
5.1 Test Amplification . . . 96
5.2 User-Guided Test Amplification . . . 97
5.3 Evaluation . . . 98
5.3.1 Design Technical Case Study . . . 98
5.3.2 Results Technical Case Study . . . 99
5.3.3 Answer to RQ1: How effective does guided test amplification generate tests for targeted branches (compared to open test amplification)? . . . 101
5.3.4 Design User Study . . . 101
5.3.5 Results User Study . . . 102
5.3.6 Answer to RQ2: How do developers perceive guided test amplification (compared to open test amplification)? . . . 105
5.3.7 Threats to Validity . . . 105
5.4 Discussion and Implications for Practitioners and Researchers . . . 106
5.4.1 Implications for Practitioners . . . 107
5.4.2 Implications for Researchers . . . 108
5.5 Related Work . . . 108
5.5.1 Directed Test Generation . . . 108
5.5.2 Interactive Test Generation . . . 109
5.6 Conclusion and Future Work . . . 109

6 Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps 111
6.1 Fuzzing To Inspire Functional Tests . . . 113
6.1.1 Inspiration Through Fuzzing-based Tests . . . 113
6.1.2 Instantiation in the Mozilla Ecosystem . . . 114
6.2 Proposing Inspirational Fuzzing-Based Tests To Developers . . . 117
6.2.1 Study Design and Execution . . . 117
6.2.2 Developer Reactions . . . 118
6.3 Selecting Relevant Coverage Gaps . . . 119
6.4 Do Developers Think These Coverage Gaps Should Be Tested? . . . 120
6.4.1 Test Relevance of the Filtered Coverage Gaps . . . 121
6.4.2 Different Ways to Address Coverage Gaps . . . 123
6.4.3 Needs of Developers and How to Improve Our Approach . . . 124
6.5 Discussion . . . 126
6.5.1 Implications for Practitioners . . . 126
6.5.2 Implications for Tool Builders and Developer Supporters . . . 126
6.5.3 Implications for Researchers . . . 127
6.5.4 Threats to Validity . . . 127
6.6 Related Work . . . 128
6.7 Conclusion and Future Work . . . 129
6.8 Appendix: Fuzzing-Discovered Security Vulnerabilities at Mozilla . . . 130

7 Conclusion 131
7.1 Revisiting Our Research Questions . . . 131
7.2 How to Design an Effective Collaboration Between Software Developers and Automatic Test Amplification Tools . . . 134
7.3 Implications . . . 135
7.3.1 Implications for Software Developers and Society . . . 135
7.3.2 Implications for Tool Builders . . . 136
7.3.3 Implications for Researchers . . . 136
7.4 Future Work . . . 137

Bibliography 141

Glossary 159

Curriculum Vitæ 161

List of Publications 163

Summary

Developer testing has become an established practice in large software projects. The developers working on the functionality of a project also write short, automated scripts that check the behavior of their code. While the benefits of developer testing are widely accepted, writing tests is still seen as tedious and time-consuming. Researchers are working towards alleviating developer effort by automatically generating tests. One approach to do this is test amplification, which modifies existing, manually written tests to create new tests that improve the strength of the existing test suite. When trying to fully automatically generate tests, test generation tools face the relevance problem and the oracle problem: Which behavior of the system is worth testing and what is the expected output to check for? The developer already needs to have an understanding of these two aspects to write the code under test. We propose to leverage this knowledge of the developer to improve the test amplification process. Conjecturing that a consciously designed interaction is the key to an effective collaboration, we propose a developer-centric approach to test amplification that uses a dedicated test exploration tool to communicate and collaborate with the developer.

During the five design science studies in this thesis we investigate key factors for an effective collaboration between software developers and test amplification tools. Starting off, we explore what is important in the amplified tests and the test exploration tool. We also study the information needs of the developer when inspecting amplified tests, and the value test amplification can provide to the developer. We explore the use of a visualization to convey the execution behavior and coverage impact of a test, and the impact of explicit guidance towards a coverage target. By collecting feedback from open source maintainers on amplified tests, we gather the types of changes developers can expect to make to amplified tests. Through discussions with developers on partial, fuzzer-based tests that target coverage gaps, we learn that not all coverage gaps are relevant to be closed or worth the effort to close them.

Concluding this thesis, we formulate guidelines on how to design an effective collaboration between software developers and test amplification tools. The insights we gained also include guidelines for developers on how to examine and edit amplified tests. We learned that developers can contribute valuable, hard-to-automate improvements to amplified tests and call to focus on supporting the developers in their contributions over further focussing on full automation. At the same time, we also observe the effort required from developers to understand and complete partial tests, which can inhibit the use of our tools. Therefore, managing the trade-off between perceived value and required effort should be central during the design of an effective collaboration between software developers and automatic generation.

Samenvatting

Het schrijven en uitvoeren van testen door software ontwikkelaars wordt in de praktijk vaak gebruikt tijdens het ontwikkelen en onderhouden van grote softwareprojecten. Dit houdt in dat software ontwikkelaars tijdens het ontwikkelen van de software ook korte, geautomiseerde scripts creëren die het gedrag van hun code controleren. De voordelen van het testen door ontwikkelaars zijn breed geaccepteerd, maar het schrijven van tests wordt nog steeds gezien als moeizaam en tijdrovend. Met deze wetenschap proberen onderzoekers het testen van software door ontwikkelaars makkelijker en minder tijdrovend te maken door automatisch testen te genereren. Een specifieke manier om dit te doen heet *test amplificatie*: bestaande, handmatige geschreven testen worden op een systematische manier veranderd om zo te leiden tot nieuwe testen die de bestaande test suite verbeteren. Echter, bij het automatisch genereren van testen worden test generatie tools geconfronteerd met het relevantie probleem enerzijds, en het orakel probleem anderzijds. Dit probleem vertaalt zich naar: welk gedrag van mijn systeem is de moeite waard om te testen en wat is het verwachte gedrag van het systeem dat we testen? De ontwikkelaar moet begrip hebben van beide aspecten om de desbetreffende code te schrijven. Wij stellen voor om de kennis van de software ontwikkelaar te gebruiken om het test amplificatie proces te verbeteren. Daarbij veronderstellen we dat een bewust ontworpen interactie tussen de software ontwikkelaar en de test amplificatie tool de sleutel is voor het genereren en voorstellen van goede geamplificeerde testen. Binnen deze visie stellen we een ontwikkelaar-centrische aanpak van test amplificatie voor die gebruik maakt van een een test exploratie tool om met de ontwikkelaar te communiceren en samen te werken.

Middels vijf studies die het design science paradigma onderschrijven bestuderen we in dit proefschrift sleutelfactoren voor een effectieve samenwerking tussen software ontwikkelaars en test amplificatie tools. We onderzoeken eerst de belangrijke aspecten van de geamplificeerde tests en de test exploratie tool. We bestuderen de informatiebehoeften van de ontwikkelaar bij het inspecteren van geamplificeerde tests, en de waarde die test amplificatie kan bieden aan de ontwikkelaar. We verkennen de toepassing van een visualisatie om het uitvoeringsgedrag en de dekkingsimpact van een test over te brengen. We bekijken ook wat de invloed is van de software ontwikkelaar die het test amplificatie proces expliciet richting een bepaald stuk functionaliteit (en dus code) duwt. We analyseren bovendien de feedback van open source onderhouders op geamplificeerde tests, verzamelen we de soorten bewerkingen die ontwikkelaars kunnen verwachten te maken wanneer zij met geamplificeerde tests werken. Op de basis van gesprekken met ontwikkelaars over gedeeltelijke, fuzzer-gebaseerde tests die coverage gaps aanpakken, leren we dat niet alle coverage gaps relevant zijn om te adresseren en te dichten.

We ronden dit proefschrift af met richtlijnen over hoe je een effectieve samenwerking tussen software ontwikkelaars en test amplification tools kan vormgeven. De inzichten die we hebben opgebouwd bevatten ook richtlijnen voor ontwikkelaars die geamplificeerde tests beoordelen en bewerken. We hebben geleerd dat ontwikkelaars waardevolle, moeilijk

te automatiseren verbeteringen aan geamplificeerde tests aanbrengen. Daarom roepen we op om meer aandacht te besteden aan het ondersteunen van de ontwikkelaar bij deze contributies in plaats van verder aandacht te besteden aan volledige automatisering. Tegelijkertijd observeren we ook de grote inspanning die van de ontwikkelaar wordt gevraagd om gedeeltelijke tests te begrijpen en te voltooien, wat het gebruik van onze tools kan bemoeilijken. Bijgevolg moet het beheren van de afweging tussen de waargenomen waarde en de vereiste inspanning centraal staan bij het ontwerp van een effectieve samenwerking tussen software ontwikkelaars en automatische generatie.

Acknowledgments

There are many wonderful people that I had the delight to meet, work, and interact with throughout my PhD. In this section, I'd like to express my gratitude and acknowledge the contributions they made to this thesis, me and my life 🙂.

Andy, it is difficult for me to put into words how immensely grateful I am for your continuous support and advice during the last years. You succeeded so well in constructively nudging me onto the right path, while letting me maintain my independent drive. I am especially thankful for your constant availability and reliability. You make time to be there when it is needed, and I learned that I can count on what you say. I particularly enjoy our conversations, in-depth reflective and thoughtful exchanges not only about our research, but to my delight also about the academic world beyond.

Arie, thank you for your spot-on advice throughout these years and giving me the confident trust and freedom to do awesome research. Beyond that, thank you for fostering an amazing research group of wonderful people to work amongst.

I want to thank my committee members, Benoit Baudry, Andreas Zeller, Xavier Devroey, Koen Langedoen, and Burcu Özkan for the careful reading of my thesis manuscript and the constructive feedback on it.

Imara, Mark, and Andra, thank you for being my best friends here in Delft. I thoroughly enjoy us hanging out, be it shooting arrows and discussing minds, deliberating theories and life, or eating Ramen and enjoying dance shows. With each of you I have someone who I can openly speak to about everything and who understands and feels with me my thoughts and struggles 🤗.

Ali, Amir, Baris, Mark, and Abir, I am grateful for you as my ever-supportive officemates and closest colleagues during this time. It is awesome to discuss your and my research ideas with each other, and that you showed me how it is to work together and collaborate with bright minds.

I want to thank all my fellow PhD Students of SERG for creating a cheerful and supportive environment for everyone around. I especially cherish that I was able to become closer friends with several of you over conference travels and other opportunities for personal conversations. Andra, thank you for our daily catchups and with that the opportunities to get our minds working again. Thank you all for showing me the many facets that doing a PhD and life beyond it have 😊.

Xavier, Maliheh, Mairieli and June, I am thankful for our conversations that were both uplifting and thought-provoking. You made me think, reflect, and grow my understanding of our world. Mauricio, Enrique and Ayushi, thank you for sparking my love for research methods and theories.

Casper, Danyao, Khalid, Nienke, Wessel, George and Swastik, thank you for the fantastic time I got to have through supervising you and collaborating together to conduct awesome research.

I am grateful to Minaksie and Kim for the joy and positivity they brought to our office and for always being there when we needed to work things out with the university.

Alberto, thank you for the chance to work together on such an awesome collaboration, for connecting me with Mozilla, and for teaching me new approaches to look at research in our field. Marco, Christian and Jason, I am grateful for the glimpse you let me have into the software and fuzzing world at Mozilla, and all the support you gave me during our project.

Thank you Walid and the whole MAST group for the warm welcome in Hamburg, showing me new facets of software engineering, academia and what university life can look like. Especially Abir, thank you for being a friend during this time and letting us explore collaborating together.

My thanks go out to the NWO for funding my position, allowing us wonderful freedom in conducting our research. I want to thank the Zonta organization and their Women in Technology Scholarship that allowed me to visit the University of Hamburg for an extended time and to broaden my horizons in research and life. I also want to thank everyone who participated in my studies, allowing me to bring out the diversity and humanity in using automatic test amplification.

I am grateful to the European Union for making it so simple to move to and live in another country, letting me experience a different culture and new perspectives on life in the same safety as back home. I am also grateful for our society's development throughout the pandemic, that we learned how to thrive with limited in-person social interaction.

Dank je wel Ron voor de ondersteuning bij het afmaken van de cover. Samen met Anita, bedankt voor een tweede familie dicht bij huis.

Dominik, Steffi und Miri, danke, dass wir Freunde geblieben sind und auch über den großen Abstand und die Pandemie immer wieder quatschen konnten.

Mama, Papa, Viki und Julia, ich bin dankbar, dass ich immer auf eure Unterstützung bauen kann und für die vielen Anstöße, doch mehr Forschung und Sport zu machen 😊.

Taico, thank you for your neverending support, open ear for everything I want to talk about, being there through highs and lows, and always being up for in-depth discussions about life 💙.

Caro

1 Introduction

Elena and her software development team are struggling with the quality of their project. Changes to the code often break existing functionality, and finding the root causes for these regressions is difficult. They know that more tests can help them prevent these errors or solve them quicker, but management is pressing them to deliver new features and gives little time for long-term quality improvement. Elena has heard of automatic test generation tools that can help them create more tests. She tries out such a tool, which promises to deliver tests that can detect many errors. When looking at the singular tests, she is not quite sure how they improve her test suite, but she adds them anyways, trusting the tool's promise. When one of the test fails after a change, she struggles to localize the underlying fault causing the failure because it is hard to understand the tested scenario. While discussing with her colleagues, they discover that the test is checking unimportant behavior which can change without impacting the users of their software. They decide to delete the generated test, and Elena is left with the feeling that the test generation was not helpful for their team.

In the established software development process, software testing is one of the pillars ensuring the quality of the produced software [1]. Software tests come in a variety of forms: from manual tests executed by humans which test the interaction with a system, over system tests which check the interaction between all components, down to automated unit tests written by developers to check the behavior of singular components [2]. In general, tests consist of an input and an expected output. The input can be defined together with instructions on how to interact with the system under test. These instructions are executed and the resulting actual output from the system under test is compared to the expected output. If the actual output matches the expected output, the test passes, if it does not match, the test fails. Depending on their form, tests can be used for a variety of purposes. For example, manual tests can be used to check the usability of a whole system and whether it fulfills the agreed upon requirements. Automatically executable system or unit tests can be used as regression tests to regularly check that the important behaviors of a software system are not altered [3, 4]. Randomized inputs can be used to

check the robustness of a system against crashes or exploits, a technique called fuzzing [5]. While it is widely known that we need strong software testing to produce high quality software, developers perceive creating or writing tests is seen as a tedious, time-consuming activity [6–9].

Automatic Test Generation

To lighten the developers' effort, software engineering researchers work on automating the generation of software tests. Just as there is diversity in the form and purposes of tests, there is a plethora of different automatic test generation approaches [10–15]. Search-based approaches use evolutionary algorithms to generate optimal test suites with regards to a given objective function, e.g., maximum coverage of the code under test with minimal length and number of test cases [16, 17]. Test amplification leverages existing manually written tests and strengthens them by mutating the input and setup and adding new assertions to increase the coverage of the test suite [13], or detect behavioral differences created by code changes [18]. Recently, machine learning and especially large language models have also been applied to generate tests [19] or to boost more traditional test generation approaches [15]. The state-of-the-art test generation tools are effective in detecting regressions [20], finding [21] and reproducing crashes [22, 23], exposing under-tested scenarios [21], and generating test data [24].

However, there are two challenges that are consistently difficult to solve for fully automatic test generation approaches:

- What behavior is relevant to test? (Relevance Problem)
- What is the correct outcome that the test should check for? (Oracle Problem)

In software engineering, developers aim to create a working software product with limited resources such as time and budget. This means that exhaustive testing of all possible inputs is not feasible, and developers need to prioritize which behavior to test [2]. Even if automated tools can generate tests for all inputs, in a regression testing scenario not all behaviors that the system exhibits are intended by the developers and necessary to preserve. In this situation, a test that fails upon irrelevant behavioral changes would unnecessarily alarm the developers, like in our initial story about Elena. We call this the *relevance problem* of test generation, relating to what input scenarios should be tested. The second challenge is the *oracle problem* [25, 26]. For each test, we need to know the intended behavior of the system under test to determine the expected output that the actual output is compared to. This is a long-standing concern in software testing [25], which has been addressed in many ways. One approach is to use implicit oracles, like the absence of crashes during fuzzing [5], deriving test oracles from known relations between input and output [27], or by relying on a formal specification of the system under test [25].

Generating Tests For and With Developers

Both the relevance and the oracle problem require a deeper understanding of the intent behind the software under test. To automatically address them we would have to analyze requirements and design documents. However, this neglects a competent resource that

already had to understand the requirements and design of the software: the software developer. To develop software, developers need to make themselves familiar with the goals that their code is supposed to fulfill [28]. This means **software developers are a valuable resource when it comes to understanding the relevance of the behavior of the software under test and the expected outcome of a test**. Our idea for this thesis is to explore how we can leverage this resource to improve automatic test generation approaches. If automatic test generation tools ask and learn from developers which scenarios to test and what the intended behavior is, would this improve the tests we can generate?

A central obstacle to incorporating the developer into automatic test generation processes is the lack of usability and developer-perceived usefulness of the current test generation tools. These tools are rather difficult to adopt in day-to-day software engineering, due to several challenges limiting the understandability of the generated tests. These challenges include, for example, readability [29, 30], meaningful names [31–33], and documentation [34–36]. This can lead to the generated tests being stored separately from the manually written tests, e.g., when the tests are regenerated after the software changed [21, 37]. We conjecture that, **for the developer to be willing to invest effort in helping our automatic test generation tools, the developers need to feel like using the tools is worth their time and effort**. To provide this value to the developer, we declare as the goal of this thesis to **work towards generated tests that software developers include in their maintained test suite** next to their manually written tests. The maintained test suite refers to the xUnit tests that developers write and maintain as part of the regression test suite of a software project [2–4]. The acceptance of a generated test into the developers' test suite is our form of a test adequacy metric, as it indicates that the developer perceives the test as relevant and useful. Such developer-accepted tests are also valuable for use cases beyond checking the behavior of code, especially for use cases that require the developer to understand the behavior and intent of the test. For example, the tests can serve as a form of executable documentation [38–40] and help developers localize the underlying fault when a test fails [35, 41].

The test generation approach called **test amplification can help us with generating tests that are easier to understand for developers**. Test amplification makes automatic changes to existing, manually written tests to generate new tests that are complementary to the existing test suite [42]. The benefit we expect from test amplification is that understanding the generated tests is easier because they are similar to the manually written tests which the developer is already familiar with. This is why we choose to use test amplification as the approach to generate tests in this thesis. In the following section, we give a more detailed introduction to test amplification and further reasons why we use this approach and the term amplification in this thesis.

Putting together our idea of leveraging the developers' insight, generating useful tests for developers and employing test amplification leads to the overarching vision for this thesis (Figure 1.1): We want to **generate amplified tests *for* developers**, tests that are useful and valuable for them. This **enables us to generate amplified tests *with* developers**, leveraging their knowledge to address the relevance problem and the oracle problem. In turn, this **enables us to generate better amplified tests *for* developers again**.

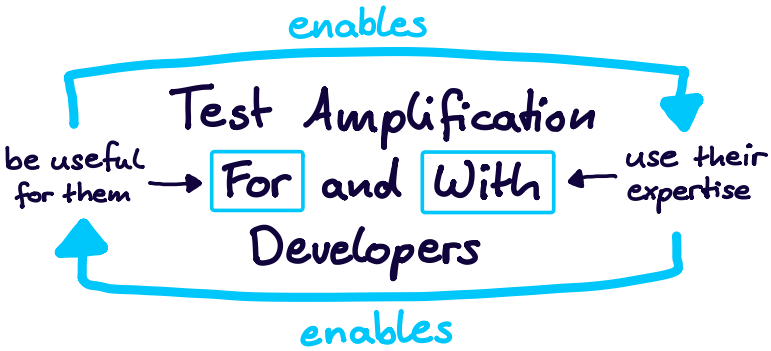


Figure 1.1: The overarching vision for this thesis.

Test Generation Through Test Amplification

In this section, we introduce the approach of test amplification and explain why we choose to use this approach throughout the thesis.

Following the definition by Danglot et al., test amplification summarizes approaches that leverage knowledge contained in existing test suites to generate new tests that improve these test suites with regards to an engineering goal [42]. The existing knowledge that is leveraged could be existing test inputs or setups. The approach generating the new or improved tests could add more assertions to strengthen the fault detection capability or modify the test execution environment to check for more unexpected behavior with the existing tests. Test amplification always targets to improve an engineering goal, such as improving coverage of the code under test, improving mutation score or uncovering behavioral changes.

The specific test amplification approach we build upon is designed by Danglot et al. [13], and implemented for Java in their tool DSpot.[1] It amplifies JUnit tests and works as shown in Figure 1.2. For the amplification, we choose an original test from the existing test suite. Then, DSpot removes all current assertions and mutates the input and setup phase of the test. These mutations could be modifying literal values or replacing them with random ones, removing method calls or adding new method calls to the objects under test. Then, DSpot generates new assertions to fit the new behavior of the mutated test. Here, the behavior of the system under test is taken as the oracle, i.e., the expected values of the assertions are chosen so that the test passes when it is executed. Finally, the new test is compared to the existing test suite. If it improves the test suite according to the chosen adequacy metric, e.g., structural coverage or mutation score, then it is kept and returned as an amplified test.

For this thesis, we choose to use test amplification to generate tests because the resulting tests are similar to the original tests. We assume that a developer familiar with the test suite will therefore more easily understand and accept amplified tests. This is also the central difference to, and potential advantage over, the widely studied and powerful search-based test generation of, for example, EvoSuite [10]. A second advantage is that test

[1] `https://github.com/STAMP-project/dspot`

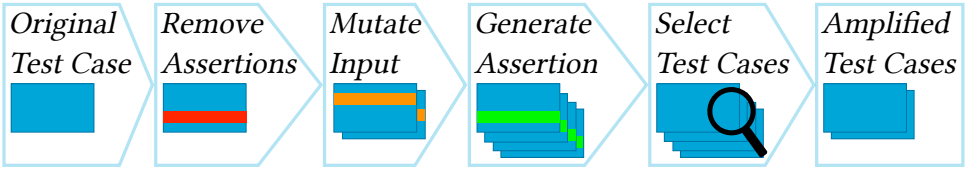


Figure 1.2: Overview of test amplification with DSpot.

amplification supports integration tests out-of-the-box. Integration tests are tests that involve more than one class under test. A large part of manually written xUnit tests are integration tests [43, 44]. The majority of the previous developer-involved studies regarding automatic unit test generation are conducted with EvoSuite [45–47] and do not consider test amplification approaches. Existing reports and efforts to improve test amplification show that it suffers from similar problems as those observed with EvoSuite, such as complex configuration and long wait times [21], as well as lacking documentation explaining the generated test [36].

We prefer and use the term test amplification because we primarily use a modification of Danglot et al.'s approach for our studies. It also emphasizes the idea of improving an existing test suite. The use case for test amplification is a developer or team wanting to improve their existing test suite. This is why the value of an amplified test is always measured in comparison to this test suite, answering the question: "How would my test suite improve if I add this test to it?" The evaluations of other automatic test generation techniques typically use a different scope to judge the adequacy of their tests, e.g., minimizing the size of a whole generated test suite while maximizing mutation score [17], or finding vulnerabilities in software systems [48].

Research Goal

The overall theme motivating this thesis is to enable test amplification to work for and together with software developers. Based on the existing developer experiences with test generation tools reported in literature, we conjecture that the crucial component towards success is the design of the collaboration between the software developer and the automatic test amplification tool. This encompasses the intended interaction workflow of these two, the test amplification process itself and the user interface facilitating the communication between developer and tool. To illustrate the potential impact of a thoughtful design of this collaboration, consider the following example of developer Elena working with the developer-centric test amplification tool and process we develop throughout this thesis:

> Elena, a software developer, wants to expand the test suite of the project she is working. The test suite should cover more functionality and allow her and her team to be more confident that they are not breaking important behavior when they make changes to the code. As her management is constantly asking for new features, she is pressed on time and decides to use an automatic test amplification tool to improve her project's test suite. From her integrated development environment (IDE), she starts the test amplification. The tool

generates several new test cases for her. Elena inspects the test cases one by one directly from the test exploration tool integrated into her IDE. Through a visualization, she observes the tests' behavior and their new coverage contribution to judge which tests to include in the test suite. When she decides to keep a test, she takes a look at the generated code and does some adjustment to make the test easier to understand for her colleagues and fit better to their project's style and quality. After adding several new tests into the test suite of her project, she commits them all and prepares a merge request that describes the improvements to the test suite.

Compared to the initial example, Elena's experience is much more positive. To create this kind of experience, we need to understand what is needed to design such a collaboration between developer and test amplification tool. Therefore, the overarching research question this thesis addresses is:

How do we design an effective collaboration between software developers and automatic test amplification tools?

To address this question, we adopt an engineering research approach, also called design science [49–51]. In engineering research, we take a proactive approach to design innovative artifacts that improve the software engineering process. We iterate between two phases: construction and evaluation. In the construction phase, we sketch and implement prototypes of our innovative artifacts based on existing knowledge and theories, as well as the needs and problems of the software engineering processes that we aim to improve. In the evaluation phase, we rigorously examine these prototypes to gain novel insights into how they affect the software engineering process and learn how to improve our and similar designs. Each chapter of this thesis, and the publications they are based on, follows these two phases: Constructing a design and a prototype, and then evaluating it in a study with software developers to gain insights into the strengths of the design and how to improve it.

Research Outline

Based on the shortcomings and recommendations discussed in the existing user studies involving test amplification and other automatic test generation methods, we develop an initial prototype of a tool that facilitates the interaction between a test amplification tool and a software developer (Chapter 2). Figure 1.3 gives an overview of the developer-centric test amplification workflow we propose, where a developer interacts with a so-called *test exploration tool.* The tool is embedded into the developer's integrated development environment (IDE) and takes care of configuring and calling the test amplification tool (③ in Figure 1.3). The test exploration tool then presents the returned amplified tests to the developer, together with information on how each test improves the coverage of the test suite (⑤). The developer can then explore the different proposed tests and judge which ones to add into their test suite (⑥ and ⑦). We implement this design in an IntelliJ plugin called *TestCube*, which leverages the original test amplification tool DSpot for Java, developed by Danglot et al. [13]. In Chapter 2 we evaluate our prototype *TestCube* through semi-structured interviews with 16 software developers. This way, we explore

which aspects to focus on when developing the idea of developer-centric test amplification further. The following chapters of the thesis each investigate one of these aspect in more depth.

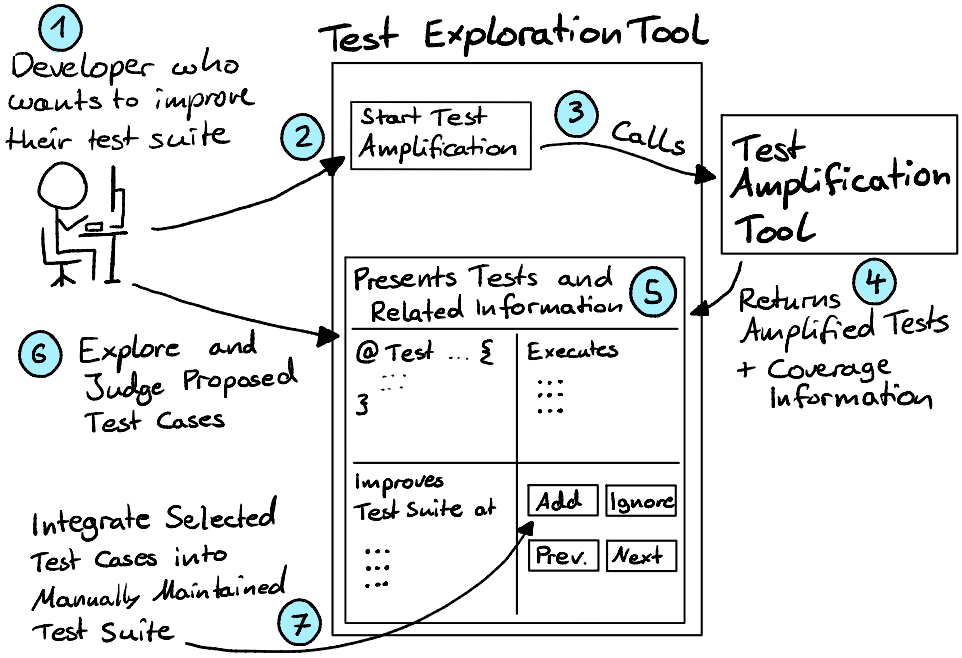


Figure 1.3: Overview of test amplification with the help of a test exploration tool.

Starting off, in Chapter 3 we develop and evaluate TestImpactGraph, a graph-based visualization that shows the methods called during the execution of an amplified test, including colored line highlights showing which lines are newly covered compared to the existing test suite. The goal of the visualization is to communicate the behavior and the coverage contribution of an amplified test, so that a developer can more effectively judge whether to include the test in their test suite.

In Chapter 4, we characterize which types of changes developers make to amplified tests before deciding to include them in their maintained test suite. For this, we first resolve issues with the test amplification identified in the evaluation of Chapter 2. We then generate amplified tests for popular open source projects, and propose them in 39 pull request to the maintainers of these open source projects. Based on our analysis of the changes requested by the maintainers, together with our own preparations necessary to obtain tests suited for pull requests, we formulate guidelines for developers. These guidelines clarify which actions they can expect when selecting and editing amplified tests before including them in their maintained test suites. Our central observation is that several of these actions highly benefit from the developers' expertise and familiarity with the software under test. Two examples of such changes are merging tests with existing tests

or providing meaningful documentation. This leads us to call for more focus on supporting developers in understanding amplified tests, so they can contribute such changes.

In Chapter 5, we explore the idea of letting the developers actively guide the test amplification towards the branches they want to cover, and study how this impacts the process of developer-centric test amplification. Our user study reveals that active guidance does help developers understand the amplified test, but that the guidance also poses new expectations in terms of being able to cover any branch, and expectations of interactive speed of the test amplification.

In Chapter 6, we divert from our prior approach and tooling for developer-centric test amplification. Together with Mozilla, we explore leveraging their established fuzzing tools to generate partial tests that, when completed by developers with a functional oracle, can close coverage gaps in their code base. Based on the reactions towards proposed fuzzing-based tests and discussions with the developers, we learn that the relevance to test a coverage gap can vary widely. The developers' intention to address the coverage gap depends on the effort to create or complete a test, the expected likelihood for a bug and how the gap is covered by other quality assurance techniques.

The research questions we address in each chapter of this thesis can be summarized as:

Research Questions For Each Chapter

Chapter 2 What factors are relevant to developers working with a developer-centric test amplification tool?

Chapter 3 How should a visualization of the execution behavior and coverage impact of an amplified test be designed to help software developers judge the amplified test?

Chapter 4 What changes do developers make to amplified tests before incorporating them into their maintained test suite?

Chapter 5 How does developer guidance towards a coverage target impact the developer-centric test amplification process?

Chapter 6 What are developers' reactions when proposing fuzzing-based tests that would close coverage gaps after being completed with a functional oracle?

Research Methods

The methods applied in the research for this thesis are primarily qualitative and exploratory. Throughout the different chapters we move between interviews and user studies in a lab-like setting (Chapters 2, 3 and 5) and studies in the wild, where we observe the reactions of developers to test amplification in their own software projects (Chapters 4 and 6). Throughout the thesis we follow the design science research methodology [49, 50], iteratively designing and evaluating prototypes of our test amplification approach or its components. For the evaluation of our prototypes, we aim for rich and actionable insights about the developers' experience and needs when interacting with test amplification tools. These

are then the basis of our results, pointing towards future steps to improve the approach and tools. Furthermore, test amplification is a relatively young approach that is not yet established in the software developers' toolbox. When the approach matures and becomes more widely used in the future, quantitative comparisons in experiments with other test creation approaches will become more applicable.

We applied the following methods in each of the chapters of this thesis:

Chapter 2 Semi-structured interviews with 16 developers while generating amplified tests with our developer-centric test amplification prototype *TestCube*.

Chapter 3 Think-aloud study with 5 developers while exploring our visualization prototype.

Chapter 4 Open source contribution study by submitting 39 pull requests with amplified tests, 36 of which the maintainers reacted to. 19 pull requests were accepted and 13 closed. Qualitative analysis of our manual pre-processing and the maintainers' change requests.

Chapter 5 Technical case study on two open source projects and user study with 12 participants applying both undirected and directed test amplification.

Chapter 6 Prototype study leading to 13 issue reports about tests, 7 of which received reactions. In addition, chat conversations with 13 developers about the relevance of closing certain coverage gaps. Qualitative analysis of feedback comments to the issue reports and the conversations.

Open Science and Reproducibility

We strive to make all our research results available to the public, which is also a requirement of the public Dutch funding agency NWO.[2] The already published papers forming the chapters of this thesis are available under green (Chapters 3 and 5) or gold open access (Chapters 2 and 6). Furthermore, each paper is accompanied by a publicly persisted artifact which provides the software we developed for our studies, raw data and analysis scripts to enable readers to validate our claims. Table 1.1 provides links to all publications and their artifacts. Research ethics and data protection were a central concern when we involved developers in research studies. For our studies with active human participants we sought approval from the human research ethics board of the TU Delft. To minimize risks and respect the rights of the participants we limited the collection of personal data to strictly necessary data and asked for informed consent.

[2] https://www.nwo.nl/en/open-science

Chapter	Open Access	Publication DOI	Artifact Link
Chapter 2	Gold	https://doi.org/10.1007/s10664-021-10094-2	https://zenodo.org/doi/10.5281/zenodo.5254869
Chapter 3	Green	https://doi.org/10.1109/VISSOFT55257.2022.00011	https://zenodo.org/doi/10.5281/zenodo.4971422
Chapter 4	Green	https://doi.org/10.1109/TSE.2024.3381015	https://zenodo.org/doi/10.5281/zenodo.7034924
Chapter 5	Green	https://doi.org/10.1109/SCAM59687.2023.00032	https://zenodo.org/doi/10.5281/zenodo.8074647
Chapter 6	Gold	https://doi.org/10.1145/3639477.3639721	https://zenodo.org/doi/10.5281/zenodo.10470823

Table 1.1: The open access status, publication links and persisted artifacts of the papers forming the chapters of this thesis.

Contributions

The research described in this thesis brings the following contributions:

1. Design of the novel method of developer-centric test amplification, where the developer can explore and judge proposed amplified tests from within their integrated development environment. (Chapter 2)

2. A qualitative interview study evaluating developer-centric test amplification, leading to insights such as the need to actively design the communication between developer and test amplification tool, as well as the importance of understandability and relevance of the test case to the developer. (Chapter 2)

3. Design of a graph-based visualization of the execution and coverage impact of an amplified test case. (Chapter 3)

4. Think-aloud evaluation of the visualization, revealing the information developers seek from such a visualization, as well as new viewpoints on tests enabled by it such as recognizing direct vs. deep coverage and the wish to see tests that cover the same code. (Chapter 3)

5. Open-source contribution study on the changes made to amplified test cases before including them in the test suite, leading to guidelines on what developers can expect as selection and editing tasks when working with amplified tests. (Chapter 4)

6. Design of directed test amplification, where the developer can guide the test amplification process towards a target branch. (Chapter 5)

7. Technical case study and user study comparing directed and undirected test amplification, eliciting tradeoffs like understandability (better for directed) or fulfillment of user expectations (better for undirected). (Chapter 5)

8. Design of a prototype to generate partial tests for uncovered code with fuzzing and proposing them in bug reports to developers. (Chapter 6)

9. Two studies on the reactions of developers to proposed fuzzing-based tests, revealing varying relevance of coverage gaps to be closed and the effort required to understand and complete partial tests. (Chapter 6)

10. Improvements to the test suites of various open source projects through tests contributed in pull requests or through proposals in issue reports. (Chapters 4 and 6)

11. *TestCube*: Developer-centric test amplification plugin for the IntelliJ IDE, supporting open and guided test amplification. (Chapters 2 and 5)

12. Improvements to the previously developed DSpot test amplification tool [13]: developer-centric amplified tests, directed test amplification, descriptions to accompany the amplified tests. (Chapters 2, 4 and 5)

Thesis Structure

The remainder of this thesis is presented portfolio-style. This means that the next five chapters primarily follow the five publications (one of which is still under revision) that the thesis is based upon. They are self-contained and can be read independently. The following publications were the basis for Chapters 2 to 6:

- Chapter 2: Carolin Brandt and Andy Zaidman. Developer-centric test amplification. *Empir. Softw. Eng.*, 27(4):96, 2022

- Chapter 3: Carolin Brandt and Andy Zaidman. How does this new developer test fit in? A visualization to understand amplified test cases. In *Working Conference on Software Visualization (VISSOFT)*, pages 17–28. IEEE, 2022

- Chapter 4: Carolin Brandt, Ali Khatami, Mairieli Wessel, and Andy Zaidman. Shaken, not stirred. How developers like their amplified tests. *IEEE Transactions on Software Engineering*, 50(5):1264–1280, 2024

- Chapter 5: Carolin Brandt, Danyao Wang, and Andy Zaidman. When to let the developer guide: Trade-offs between open and guided test amplification. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 231–241. IEEE, 2023

- Chapter 6: Carolin Brandt, Marco Castelluccio, Christian Holler, Jason Kratzer, Andy Zaidman, and Alberto Bacchelli. Mind the gap: What working with developers on fuzz tests taught us about coverage gaps. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2024

The final chapter of this thesis, Chapter 7, summarizes and connects the findings from the chapters and discusses future work.

2

Developer-Centric Test Amplification: The Interplay Between Automatic Generation and Human Exploration

Automatically generating test cases for software has been an active research topic for many years. While current tools can generate powerful regression or crash-reproducing test cases, these are often kept separately from the maintained test suite. In this chapter, we leverage the developer's familiarity with test cases amplified from existing, manually written developer tests. Starting from issues reported by developers in previous studies, we investigate what aspects are important to design a developer-centric test amplification approach, that provides test cases that are taken over by developers into their test suite. We conduct 16 *semi-structured interviews with software developers supported by our prototypical designs of a developer-centric test amplification approach and a corresponding test exploration tool. We extend the test amplification tool DSpot, generating test cases that are easier to understand. Our IntelliJ plugin* TestCube *empowers developers to explore amplified test cases from their familiar environment. From our interviews, we gather* 52 *observations that we summarize into* 23 *result categories and give two key recommendations on how future tool designers can make their tools better suited for developer-centric test amplification.*

This chapter is based on *C. Brandt and A. Zaidman. Developer-Centric Test Amplification, Empirical Software Engineering, 2022* [52].

Testing is an important [1], but time-consuming activity in software projects [6–8]. Automatic test generation aims to alleviate this effort by reducing the time developers spend on writing test cases. The software engineering community has created a plethora of powerful tools, that can automatically generate JUnit test cases for software projects written in Java. For example, a widely known tool is EvoSuite [10], which generates test cases from scratch using search-based algorithms. It starts from a group of randomly generated test cases and optimizes them by mutating their code and combining them with each other. This chapter focuses on *test amplification*, a technique that automatically generates new test cases by adapting existing, manually written test cases [42]. The state-of-the-art test amplification tool DSpot [13] mutates the setup phase of manually written test cases and generates new assertions to test previously untested scenarios. For both EvoSuite and DSpot, studies have shown that the tools are effective in generating or extending test suites to reach a high structural coverage and mutation score [10, 13, 47, 57].

Automatic test generation is, for example, used to detect regressions [20], reproduce crashes [22, 23], uncover undertested scenarios [21] and generate test data [24]. For these use cases, it is often sufficient to keep the generated test cases separate from the manually written and maintained test suite [21, 37]. This separation is reinforced by several hard-to-solve challenges that limit the understandability of the automatically generated tests, such as their readability [29, 30] or generating meaningful names [31, 32], or documentation [34–36].

The amplified test cases created by DSpot are closely based on manually written ones. This opens up the chance to generate test cases that are easier to understand by developers, as they are likely familiar with the original test case, which the amplified test case is based on. In this chapter, we want to leverage this aspect. We take a look at generating amplified test cases that developers can take over into the manually maintained test suite as if they would have written them themselves. To describe this kind of test generation we use the term *developer-centric*, as the developer accepting the test case is central for this kind of test generation:

> Test amplification is *developer-centric* if it aims at generating test cases that are accepted by the developer and taken over into the manually maintained test suite.

Generated test cases that are accepted by developers and are part of the maintained test suite also fulfill several further typical uses for developer tests. For example, as a form of executable documentation [39, 40, 58], or to locate the fault that causes a failing test by understanding the test in question [35].

To provide amplified test cases that developers take over into their test suite, the interaction of the developer with the test amplification tool in which they review the proposed amplified test cases is critical. In past projects, users of DSpot reported that the tool was complex to configure and they had to wait long for the tool to finish and for them to see results [21]. There is little support that guides developers through the list of generated test cases so they can effectively judge whether to keep or discard a newly amplified test case. To address these issues and realize developer-centric test amplification, we embed the test amplification tool in a so-called *test exploration tool*:

A *test exploration tool* forms the interaction layer between the developer and the test amplification tool. It lets the developer start the test amplification tool, and later explore and inspect the different amplified test cases.

To illustrate how a developer would use a developer-centric test amplification approach with a test exploration tool, we introduce an exemplary use case, which is also illustrated in Figure 1.3:

Hannah, a software developer, wants to expand the test suite of the industrial project she is working on to cover more functionality and give her confidence that they are not breaking important behavior when changing something. As her management is constantly asking for new features, she is pressed on time and decides to use an automatic test amplification tool to improve her project's test suite. From her integrated development environment (IDE), she starts the test amplification. The tool generates several new test cases for her and notifies her that it is finished. Hannah inspects the test cases one by one directly from the test exploration tool integrated into her IDE. The exploration tool shows her the code of each new test case and where in the production code new instructions are covered. Hannah browses through the new test cases and if she is happy with any test case, she adds it to the test suite with one click of a button. After exploring all proposed test cases, she commits her changes and can lean back with the confidence of a better tested system.

In this chapter, we investigate how we should design a developer-centric test amplification approach to be successful with developers. As automatic test amplification is not widely used and to prevent re-studying the already known issues in current tools, we develop prototypes of a developer-centric test amplification approach and a corresponding test exploration tool. To motivate the design choices we take for our prototypes, we derive four design intentions from the test generation literature and the use case we propose for developer-centric test amplification. Based on these intentions we revise the test amplification process of DSpot to generate shorter, easier to understand test cases. Our corresponding IntelliJ plugin *TestCube* lets the developer generate and explore test cases right from their integrated development environment (IDE).

We conduct a qualitative study to explore which aspects of our prototypes are successful in supporting developer-centric test amplification and uncover further aspects that should be addressed to realize it.

In previous studies, developers using EvoSuite were "concerned about the readability of generated unit tests, the generated input data, and the generated assertions" [45], while DSpot users found it difficult to understand the generated test cases [21]. Stemming from these observations, we investigate in our first research question what developers find important in the code and behavior of an amplified test case. The answers to this question give guidance on what factors in amplified test cases are relevant for developers to include the test cases in their maintained test suite.

Research Question 1

What are the key factors to make amplified test cases suited for developer-centric test amplification?

In our second research question, we explore how test exploration tools should be designed to support developer-centric test amplification.

Research Question 2

What are the key factors to make test exploration tools suited for developer-centric test amplification?

A powerful capability of such test exploration tools is to provide the developer with information beyond the test code itself. We already know from test review that developers are interested in understanding the code under test, as well as knowing the code coverage of test cases [59]. To deepen our insights into what information test exploration tools should make accessible to developers, we pose our third research question.

Research Question 3

What information do developers seek while exploring amplified test cases?

Creating a great tool alone is not enough for developers to appreciate using it. We also need to convey the value our tool brings to them. Therefore our fourth research question asks what value developers can gain from using developer-centric test amplification. With the answers to this question, future tool creators know which benefits they can focus on when they seek to convince users to start or keep using their tool.

Research Question 4

What value does developer-centric test amplification bring to developers?

To answer these research questions, we conduct semi-structured interviews with 16 software developers from varied backgrounds. The participants tried out our prototypes and provided us with rich insights on their impressions of our prototypes and how we could improve them to better fit their needs. We group 52 recurring observations from the interviews in 23 result categories for our four research questions. During the discussion of these results, we identify two key recommendations on how we should design future developer-centric test amplification tools.

With this chapter, we are taking a step towards developer-centric test amplification. In short, we contribute:

- two recommendations on how to design developer-centric test amplification tools
- a structured overview of the key factors to make amplified tests as well as test exploration tools suited for developer-centric test amplification

- a refined, developer-centric test amplification approach, based on the DSpot test amplification
- a developer-oriented test exploration plugin for the IntelliJ IDE

2.1 Creating Developer-Centric Test Amplification

In this chapter, we aim to investigate the key aspects that make amplified test cases and test exploration tools suited for developer-centric test amplification by conducting semi-structured interviews with software developers. To illustrate the concept of test amplification to our participants and receive rich and concrete input, we want to let them try out a test amplification tool during the interview. A state-of-the-art test amplification tool is DSpot [13], which was developed and evaluated during the European H2020 STAMP project. We adapt DSpot's amplification process based on the feedback from the reports of the European project [21] and the requirements posed by our use case of developer-centric test amplification. To facilitate the interaction of the developer with the test amplification we also design a prototype of a test exploration tool.

In this section we discuss the inspirations leading to our design of both prototypes, which is described in Section 2.2. The goal of this section is to clarify our reasoning behind the design choices we took and connect them to the existing literature and user reports about DSpot. We formulate four design intentions and present their connection to our design choices in Table 2.1.

A central part of the load on developers comes from them having to understand the test cases and judge whether they check intended behavior. According to Meszaros, obscure tests that are difficult to understand at a glance are an anti-pattern, as it makes tests harder to maintain and potential bugs in the test code more difficult to detect [3]. As automatically generating human-readable code is a hard problem to solve, readability and understandability of generated test cases are recurring topics in developer's feedback: developers from the STAMP project stated about DSpot that it was hard to interpret the tool's output and the "resulting tests were difficult to understand for a human developer" [21]. In some cases, the developers found the generated tests useful, but so hard to read that they wrote a corresponding test case themselves. Nevertheless, they were glad that DSpot pointed to real bugs and supported them in testing exceptional behavior in systems where only the optimal behavior was tested before. Several previous works in test generation were concerned with making the generated tests more understandable for developers [32, 34, 35, 47, 60]. This clearly shows that we should also take understandability into account while designing our prototypes. Therefore, our first design intention is to generate test cases that are understandable for developers.

Intention 1 (I#1)

Generate test cases that are understandable for developers

From literature and our own experiences, we understand that the users of current test amplification tools face obstacles that lead them to abandon automatic test amplification. During the STAMP project [21], various industrial partners noted that DSpot takes very

long to generate test cases. The configuration is overly "complex because of the multitude of possible parameter values" which require experience to tweak correctly. The high effort required by users was reported for other test generation tools, too. Previous studies of EvoSuite pointed out the high load on developers to inspect generated test suites and to decide if assertions in test cases are correct [46]. They also spend a lot of time analyzing generated test cases to decide whether to improve or discard them [46] as generated test cases tend to be less readable than manually written ones [30]. That is why another intention leading the design of our prototypes is to decrease the load on the developers while they use test amplification.

Intention 2 (I#2)

Easy interaction to decrease the load on the developer

Another intention leading our design, is that the amplification process should be fast enough so that developers can start it and receive new test cases in the same session. This means, for example, that they do not have to wait for an external build process to finish. We conjecture that this makes it easier for them to understand the results and the value of the test amplification as they can include it directly while they work on improving their test suite.

Intention 3 (I#3)

Fast enough for direct interaction

Lastly, it is our intention that the developers can grasp the impact an amplified test case. They should see the test case as a useful addition when taken over into their test suite. Impact in this case could refer, for example, to the coverage, code quality, test code size or test suite runtime. We assume that understanding the impact is a pre-requisite to deciding whether the test is useful or not. The test exploration tool should make the impact and the quality of the amplified test cases clear so that the developers see the value that the automatic test amplification brings them.

Intention 4 (I#4)

Impact is clear to developers and they find the tests useful

Both (I#2) and (I#4) can not be addressed by modifying the test amplification of DSpot itself. Rather this shows the need for a layer in between the test amplification tool and the developer that facilitates their interaction. This role is taken by the test exploration tool.

2.2 Bringing Test Amplification to the Developer (IDE)

Based on the intentions we defined in Section 2.1, we develop prototypes for both a developer-centric test amplification tool as well as a test exploration tool. We will use these

	Design Choices															
	Test Generation									Test Exploration						
Intention	Amplification (C#1)	Remove calls inside old assertions (C#2)	One input mutation (C#3)	Explanatory comments (C#4)	One assertion generated (C#5)	Assertion matching input (C#6)	Instruction coverage (C#7)	Look for additonally covered inst. (C#8)	Report coverage (C#9)	IDE plugin (C#10)	Start with run (C#11)	Background task (C#12)	Default configuration (C#13)	Coverage information text (C#14)	Coverage editor (C#15)
Generate test cases that are understandable for developers (I#1)	x	x	x	x	x			x							
Easy interaction to decrease the load on the developer (I#2)	x									x	x	x	x		
Fast enough for direct interaction (I#3)							x								
Impact is clear to developers and they find the tests useful (I#4)						x			x					x	x

Table 2.1: The relation between our design intentions (I#1-4) and the design choices we take for our developer-centric test amplification and exploration prototypes (C#1-15).

prototypes during our interviews to illustrate a possible version of developer-centric test amplification. In the following, we present our design and explain how our choices are motivated by the intentions we set. Table 2.1 gives an overview of these choices, which we mark throughout the text with (C#n).

This section starts with a more detailed definition of test amplification and clarifies why we choose to base our developer-centric test generation on this technique. We describe how we adapt DSpot to generate test cases that are better suited to be read by developers. Further, we present our test exploration tool, the IntelliJ Plugin *TestCube*, which enables developers to easily use our developer-centric test amplification with minimal configuration, right from their familiar development environment.

2

Introduction to Test Amplification Test amplification is a term for test generation techniques that take manually written test cases as their primary input. Danglot et al. conducted a literature study to map this emerging field and defined test amplification as follows:

> Test amplification consists of exploiting the knowledge of a large number of test cases, in which developers embed meaningful input data and expected properties in the form of oracles, in order to enhance these manually written tests with respect to an engineering goal (e.g., im- prove coverage of changes or increase the accuracy of fault localization). [42]

For our prototype design we choose test amplification to generate the test cases. We exploit the existing test cases, as well as the code under test to create additional test cases that improve the instruction coverage of a test suite.

We base our approach on test amplification (C#1), because we expect that for a developer already familiar with the test suite it will be easier to understand a variation of an existing test case than a completely new one. In addition, most software projects that are looking to improve their testing already have at least a rudimentary test suite.

2.2.1 Developer-Centric Amplification With DSpot

During our interviews, we want to showcase a possible version of developer-centric amplified test cases to software developers. We adapt Danglot et al.'s tool DSpot [13], addressing the issues which were already reported by developers. Figure 2.1 gives an overview of our revised test amplification approach. Starting with the original test case from the existing test suite, we remove all existing assertions, modify the objects and values in the setup phase of the test case, add new assertions based on the changed behavior, and select test cases that cover additional instructions in the code under test.

Our design and implementation is strongly based on Danglot et al. [13] and DSpot version 3.1.0[1]. We created a fork of their repository[2] and contributed our changes back to DSpot through an accepted pull request[3]. In the following, we describe for each step the behavior of the original amplification as well as the changes we made to generate more understandable test cases (I#1) and convey their value to developers more easily (I#4). We illustrate our explanations with a running example in Figures 2.2, 2.3, 2.4 and 2.5.

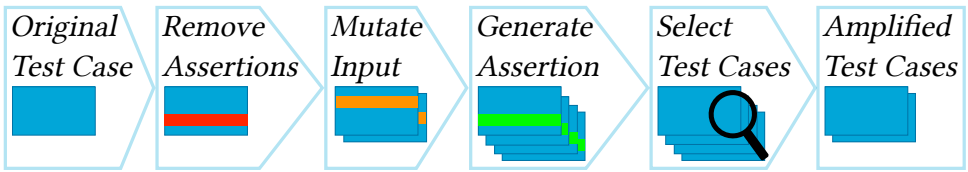


Figure 2.1: Overview of our automatic, developer-centric amplification process within DSpot.

[1] https://github.com/STAMP-project/dspot/releases/tag/dspot-3.1.0
[2] https://github.com/TestShiftProject/dspot/releases/tag/v3.2.0-dev-friendly
[3] https://github.com/STAMP-project/dspot/pull/993

```
public class AttributeTest {
    @Test
    public void html() {
        Attribute attr = new Attribute("key", "value &");
-       assertEquals("key=\"value &amp;\"", attr.html());
-       assertEquals(attr.html(), attr.toString());
    }
}
```

Figure 2.2: Example amplification: Remove all existing assertions from the original test case.

```
public class AttributeTest {
    @Test
    public void html() {
-       Attribute attr = new Attribute("key", "value &");
+       // FastLiteralAmplifier: change string from 'value &' to 'Hello\nthereNBSP'
+       Attribute attr = new Attribute("key", "Hello\nthereNBSP");
    }
}
```

Figure 2.3: Example amplification: Mutate string parameter in the constructor of the object under test.

Remove Assertions

At the start of the amplification process, DSpot removes all assertions in the original test case, as they will likely no longer match the new amplified test case. All method calls within the assertions are preserved because they might have side effects that influence the rest of the method calls in the test case.

However, these method calls tend to be confusing outside of the context of the assertion. As the behavior of the test cases is changed through the amplification anyways, we decide to also remove method calls within assertions (C#2). Figure 2.2 shows the first amplification step for our example, where the two assertions of a test case are removed completely.

Mutate Input

DSpot uses a variety of mutations to explore the input space of a test case. Literals like integers, booleans, and strings are slightly modified or replaced by completely random values. On existing objects, the input amplification removes, duplicates, or adds new method calls. It can also create new objects or literals that are then used as parameters for mutated method calls.

Our developer-centric amplification leverages the powerful input mutations of DSpot. However, from Grano et al. we know that the more complex a test case is, the harder it is to understand for a developer [61]. To make the generated test case easier to understand for the developer, we focus on one input modification at a time (C#3) and add an explanatory comment to every mutation (C#4). We make use of all available mutation operations in DSpot 3.1.0. In Figure 2.3 one of the string parameters in the constructor for the object `attr` is replaced with a new string that contains the special *non-breaking space* character. We also add a comment that details which value was changed to which new value to help the developer spot the change easily.

2

```
public class AttributeTest {
    @Test
    public void html() {
        // FastLiteralAmplifier: change string from 'value &' to 'Hello\nthereNBSP'
        Attribute attr = new Attribute("key", "Hello\nthereNBSP");
+       Assertions.assertEquals("key=\"Hello\nthere \"", ((Attribute)
  ↪ (attr)).toString());
    }
}
```

Figure 2.4: Example amplification: Generate an assertion which checks a behavior changed by mutating the input.

```
Amplified test case 'html_literalMutationString19_assSep92'
This test case improves the coverage in these classes/methods/lines:
org.jsoup.nodes.Entities:
escape
L. 197 +3 instr.
L. 198 +5 instr.
```

Figure 2.5: Example amplification: Information about the coverage improvement of the amplified test case.

Generate Assertion

Generating new assertions is one of the central features of DSpot. The tool instruments the test case to observe the state of the objects under test after the setup phase. Then it generates assertions comparing the return value of every method call on the objects under test with the observed value. While adding all generated assertions leads to a more powerful test case with respect to structural coverage, it also makes the test case hard to understand and unclear which of the added assertions improve these metrics. To minimize the generated test cases, DSpot provides a `prettifier` stage. It removes the assertions one by one, reruns the metric calculation, and adds the assertion back if the score decreased. Unfortunately, the stage multiplies the already long runtime of DSpot.

To generate shorter, more understandable test cases (I#1), we opt to only add one assertion to each test case (C#5). While this at first generates more test cases, the ones with assertions that do not improve the final selection metric are excluded in the following step. To produce test cases that developers find useful (I#4), the generated assertion should assert a behavior that changed through the previously mutated input. To achieve this, we compare all assertion candidates before and after the mutation and only include an assertion if the value it asserts changed through the mutation (C#6).

As shown in Figure 2.4, the assertion generated for our example checks the return value of `attr.toString()`, which shows the changed input `"Hello\nthereNBSP"`.

Select Test Cases

After generating a broad range of test cases through mutating input values and generating assertions, DSpot selects which test cases to keep. Depending on the configuration, DSpot selects test cases that improve instruction coverage, improve mutation score, or cover the changes in a specific commit.

As determining the mutation score is computationally expensive, it is currently not a feasible option if the test generation should run on the developer's local computer and we want to enable direct interaction with as little wait time as possible (I#3). Therefore, we select test cases based on instruction coverage (C#7).

DSpot originally keeps all generated test cases that by themselves cover more lines than the original test case they are based on. However, for a developer, it is not important that the coverage of one test case is high. Rather, a new test case should contribute *additional coverage* to the test suite. To determine this, we measure the instruction coverage of the original test suite on a fine-grained, line-by-line basis. For each generated test case, we check whether it covers *additional* instructions on any line (C#8). If that is the case, we keep the test case, if not, we discard it. The combination of this fine-grained coverage comparison together with the small number of additions we make to the original test case (C#3) (C#5) enables us to generate smaller test cases (I#2) compared to DSpot. Furthermore, these test cases have a local and therefore easier to understand impact on the coverage of the test suite (I#4).

To communicate to the developer which additional instructions are covered, our developer-centric amplification reports for each test case in which lines in the production code additional instructions are covered (C#9). Figure 2.5 shows a pretty-printed version of the additional information we provide. The amplified test case in our example covers 8 more instructions over two lines in the `escape` method of the `Entities` class.

2.2.2 Test Exploration Plugin *TestCube*

For successful developer-centric test amplification, we conjecture that the second important step to support developers with amplification test cases is *exploration*. In this section, we describe the design of our prototype of a developer-centric test exploration tool: *TestCube*. To make our new, powerful test amplification easily accessible to developers (I#2), we develop *TestCube* as a plugin to the IntelliJ IDE (C#10). A lot of previous research points out the importance of integrating tools into existing development environments. It reduces time and focus lost by switching tools [62] and enables developers to inspect test cases and related code [59]. *TestCube* is open source[4] and available to download on the JetBrains Marketplace[5]. The screenshot in Figure 2.6 illustrates the user interface of *TestCube* and how a developer would interact with it. In the following, we present how developers can use *TestCube* to amplify tests right from their editor, inspect the generated test cases and easily integrate them into their code base.

Starting the Amplification

After installing *TestCube*, the developer can start the amplification in the same way as she would execute a JUnit test (C#11). She picks an original test case to be the input for the amplification process (① in Figure 2.6). Then she can click on the green arrow next to the test, and select the new option 'Amplify' (② in Figure 2.6). After she selects this option, *TestCube* starts amplifying the test in a background task (C#12).

For other test generation tools, research has shown that even though tuning parameters to the specific project increases performance, using the default settings already produces good results [63]. To take the configuration burden off the developer as much as possible (I#2) during our interviews and to evaluate how well our new approach performs with its default configuration, we choose to set default values for the vast number of DSpot parameters (C#13).

[4] `https://github.com/TestShiftProject/test-cube`
[5] `https://plugins.jetbrains.com/plugin/14678-test-cube`

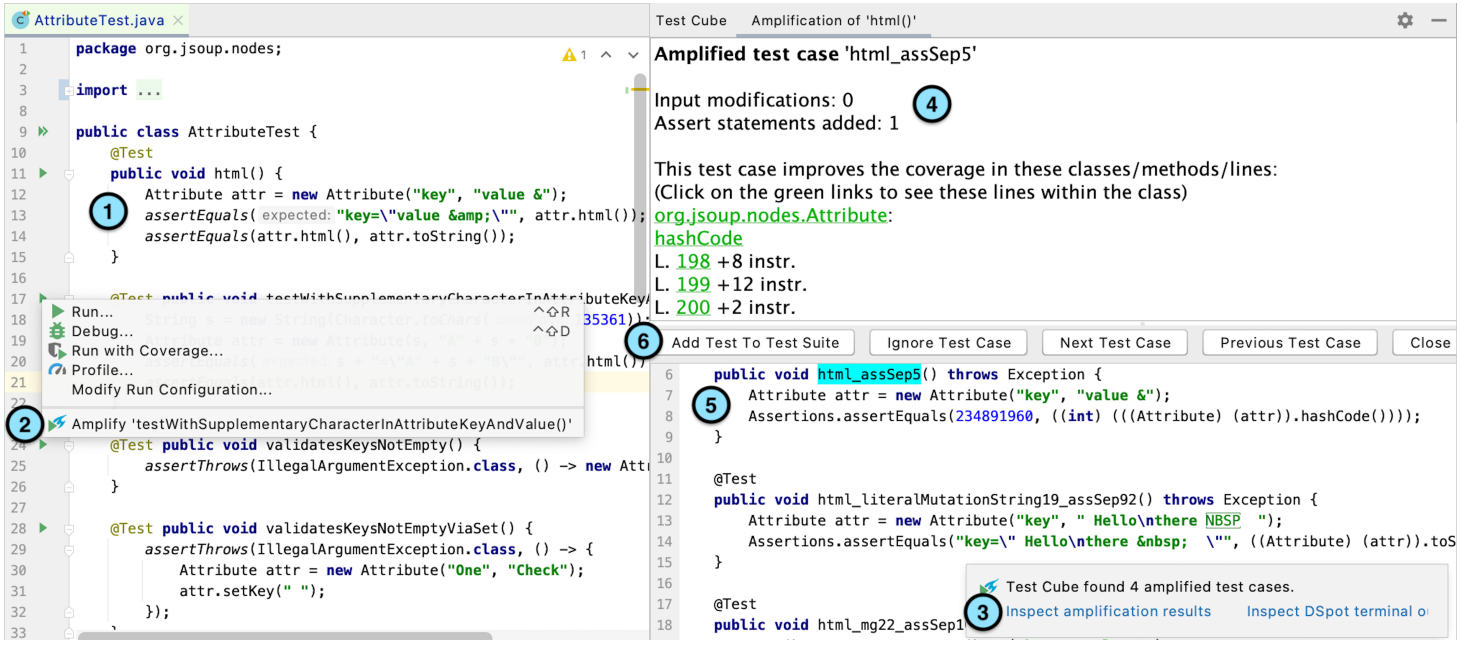


Figure 2.6: Overview of the interaction with the *TestCube* plugin.

Result Inspection

When the amplification finishes, *TestCube* notifies the developer with a pop-up from the built-in notification system (③ in Figure 2.6). The developer can then choose to inspect the test cases or, in case of reported errors, the terminal output of DSpot.

We present the results of the test amplification within IntelliJ, but in a tool window separated from the code. The tool window is located on the right, next to the editor with the original test case selected by the developer. It has various components:

- At the top of the tool window we present information about the currently selected amplified test case (④ in Figure 2.6): Which modifications were applied to it and which additional instructions are covered (C#14).

- Next we present *navigation buttons* to the developer (⑥ in Figure 2.6). With these buttons, the developer cycles through the proposed test cases, can add the current test to the test suite, ignore the current test or close the amplification results all together.

- Below the navigation buttons we present the amplified test cases in a fully functioning editor, as shown in (⑤ in Figure 2.6). The developer can edit the test case in their familiar environment and use code navigation to inspect called methods.

- The developer can click on the additionally covered lines in the *test case information* to open up the *coverage inspection editor* on the bottom of the tool window.

  The editor shows the class where coverage was improved and highlights all additionally covered lines in green (C#15). Showing the added coverage in the context of the code under test should help the developer judge the value of the generated test case (I#4).

2.3 Study Design

The goal of this chapter is to explore which aspects are important to create a successful developer-centric test amplification approach. To this end, we invited 16 software developers to try out our prototype *TestCube* on an example project. We observed their interaction with the tool and interviewed them on their experience and opinions on how an ideal test generation tool should behave. We report our observations split along four sub-questions: With the interviews we investigate what makes amplified tests (**RQ1**) and exploration tools (**RQ2**) suited for developer-centric test amplification, what information the developers are seeking while investigating the test cases (**RQ3**) and what value developers see in test amplification (**RQ4**). In the following we describe the design of our interview study: How we recruit participants and ask for their consent, our technical setup, the flow of one interview as well as our data collection and analysis process.

2

Research Questions

RQ1: What are the key factors to make amplified test cases suited for developer-centric test amplification?

RQ2: What are the key factors to make test exploration tools suited for developer-centric test amplification?

RQ3: What information do developers seek while exploring amplified test cases?

RQ4: What value does developer-centric test amplification bring to developers?

2.3.1 Preparation

We used convenience sampling to recruit participants for our interviews. We posted about it on Twitter[6] and wrote to existing industry contacts. In addition, we contacted participants of a previous survey about motivation to write test cases who indicated to be open for a follow-up interview.

As we are conducting a study with human participants, we followed the guidelines of the TU Delft's Human Research Ethics Council[7] and submitted our study design to them for review. Before each interview, we explained to the interviewees how we will process their data and asked for consent on participating in the interview, recording the session for later analysis and publishing the anonymized results in an online research repository. One participant wished to not be quoted and the corresponding results not to be published in a research repository, therefore our online appendix [64] excludes the data collected from that interview.

To showcase *TestCube* to our participants, we selected the open-source HTML parser *jsoup*[8]. Jsoup is a mid-sized Java project (35K lines of code), which is built with Maven, tested with JUnit 5 and was part of Danglot et al.'s evaluation of DSpot [13]. We chose it because we expected HTML to be a relatively simple and widely understood application domain, which would require less time to explain to our participants during the interviews. Jsoup has a test suite with a relatively good instruction coverage of 86 %. Our interviews focused on the classes `Attribute` and `AttributeTest`, as we expected the concept of an HTML attribute to be known by our participants. `Attribute` is fairly well tested, with most functions covered by the test suite. However, its custom implementation of `hashCode`, `clone` and several branches in `equals` were not covered.

To take the setup burden off of our participants, we set up an instance of IntelliJ with our plugin on a server and let the interviewees interact with it through the browser. This was possible through the JetBrains Projector tool[9].

[6] `https://twitter.com/laci_noire/status/1328334375537299461`, showed to 9527 users, 406 interactions
[7] `https://www.tudelft.nl/over-tu-delft/strategie/integriteitsbeleid/human-research-ethics`, last visited March 1st, 2021
[8] `https://github.com/jhy/jsoup`
[9] `https://jetbrains.github.io/projector-client/mkdocs/latest/`

2.3.2 Interview Procedure

To get a rough context of the participant's opinion on and knowledge about software testing, we asked an open question about the participant's prior experiences with testing software. Then we briefly introduced test amplification: automatically modifying existing test cases to generate new ones that improve the coverage and can be taken over into the test suite. We explained that the goal of the interview is to see their interaction with our tool and gather feedback on what aspects they like, what they would change, and how they would use such a test amplification tool in their work. We sent the developers a link to our online setup of IntelliJ which they accessed through their browser. We introduced the example project and explained how to start *TestCube*. From this point on we invited the interviewee to explore on their own, thinking aloud about all their thoughts and impressions. We did not define an explicit task to solve, rather our introduction of test amplification and the user interface of *TestCube* animated the participants to browse through the test cases and judge whether to include them, and in some cases include them in the test suite of the example project. We kept any more explanations to a minimum to observe a situation as close as possible to the developer interacting with the tool alone. We let each participant amplify and browse through several test cases for about twenty minutes. During this time we ask them to think aloud about their impressions. We nudge them by asking questions about their actions and opinions on *TestCube*'s behavior. At the end we asked them to fill out the *System Usability Score* questionnaire, a metric frequently used in the field of Human-Computer Interaction to assess how useable a product is [65]. While filling out the questionnaire, we ask them to reflect on the usability of the plugin and how it could be adapted to better fit their needs.

2.3.3 Data Collection and Analysis

We recorded all interviews, including the screen of the developer while they were interacting with *TestCube*. In addition, the interviewer took extensive notes. We performed open coding [66] to analyze the interview notes, checking back with the recording when anything was unclear or missing from the notes. Following that, we applied axial coding [66] to structure the emerged codes. We report our findings along these axial codes, which we assigned to each of the research questions. Table 2.2 presents the axial codes arising from our analysis of the interviews.

All interviews and the initial coding were performed by the first author. To increase the validity of our analysis, the second author watched two of the performed interviews, took notes and coded them separately. Then we compared the codes both authors created for the validation interview and refined our coding schema and our interpretations of the interviews. We saw that both focused on different aspects of the interviews, one assigning about 20 codes and the other one about 10 codes per interview. In total, we agreed on 90 % of the assigned codes in the first validation step. As a second validation step, we performed an inter-rater reliability analysis. We selected re-occurring topics from our codes that appear in 4 or more interviews. The second author assigned them to 3 further interviews. To compare the assignments of both authors, we calculate the percentage agreement (70 %) and Cohen's Kappa (60 %, moderate agreement). The value of Cohen's Kappa is relatively low, because some of the codes we validate have skewed values. If a code appears in nearly all the cross-validated interviews, its chance of appearance is close to one, leading to a

small Cohen's Kappa because arithmetically the agreement could be a coincidence. We provide our code book together with the code's frequencies in the interviews, as well as our cross-validation ratings in our online appendix [64].

2.4 Results

In this section, we present the results we elicited from our study. Firstly, we give an overview of key demographic factors characterizing our study participants. Secondly, we detail what factors are important to the developers when it comes to the generated test cases themselves (**RQ1**) and which aspects make a test exploration tool developer-friendly (**RQ2**). Next, we describe the various kinds of information the developers sought while exploring the generated test cases (**RQ3**) and what value our interviewees saw in automatic test generation (**RQ4**). Every *observation* that comes from the interviews will be labeled with (O#n) and if it is directly tied to one of the codes we assigned, we also report its *support*, i.e., in how many interviews we observed it. The observations without explicitly mentioned support summarize multiple codes, describe anecdotal evidence or report general impressions we obtained overarching the single interviews. Even though we cannot link them to a specific code from our interview notes, we still consider them valuable to report for our qualitative study.

While high support signals that a topic is very relevant for our participants, we cannot infer from a small support number that an aspect is less relevant. As we wanted to explore as many aspects of developer-centric test amplification as possible, we mainly let the comments of our participants guide the direction of the interviews, similar to an unstructured interview [67]. This lead to many observations only appearing in a small number of interviews, possibly because the topic they concerned was only reached in a small number of interviews.

Throughout this section, we structure our explanation along the axial codes of our results presented in Table 2.2.

2.4.1 Participants

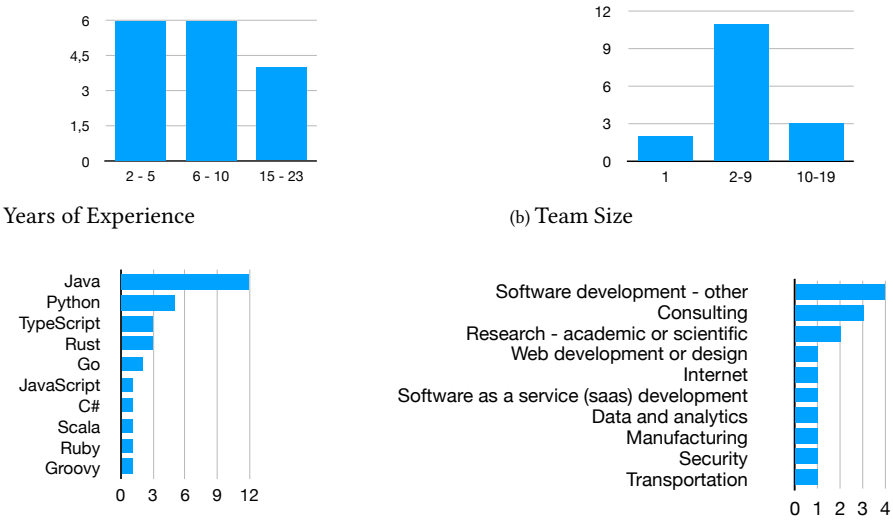
We recruited 16 participants for our study, whose demographics are summarized in Figure 2.7. As shown in Figure 2.7a, their previous experience with software development was distributed in a range from two to 23 years. Most of them work in teams of two to nine and consider Java to be among their primary programming languages. Our participants work in a wide variety of industries, which are presented in Figure 2.7d.

2.4.2 RQ1: What Are the Key Factors to Make Amplified Test Cases Suited for Developer-Centric Test Amplification?

During the interviews, it quickly became clear to us how important the test cases themselves are for the developers. Many of our participants were directly focusing on the test cases and spend much of their time praising or critiquing them (O#1). This is reflected in the large number of observations with high support we present in this section. What unified our participants is that they tried to understand the generated test cases (O#2). Rojas et al. also noted that how easy it is to obtain the intent and behavior of a test case is a strong indicator for its quality [47].

Generated Test Cases **RQ1**	Code	Identifiers Concise Consistent
	Behavior	Relevant Invariant Diverging
Exploration Tools **RQ2**	Ease of Use	Minimal Configuration Integration Usability
	Information Management Focus	
	Expectation Management	Runtime Capabilities
Sought Information **RQ3**	Test Case	Behavior / Intent Outcome Runtime
	Code Under Test Coverage Original Test Case	
Value for Developers **RQ4**	Improve Test Suite	Ease Test Engineering Inspiration
	Learning Confidence	

Table 2.2: Structured overview of the answers to our research questions. Shown are the axial codes we obtained during our data analysis described in Section 2.3.3.



(a) Years of Experience

(b) Team Size

(c) Primary Languages

(d) Industry Domain

Figure 2.7: Summarized demographics of our interview participants.

Code: Identifiers Looking at the **code** of the test cases, the most prevalent comment was the need for less cryptic **identifiers** (O#3, Support: 14). The developers wished that the identifiers, such as the test name or variable names, convey the intent of the new test case (O#4, Support: 2). They gave examples such as which code is additionally covered or simply which methods are newly called. More expressive identifiers would help them understand the intent of the test case faster. While integrating the test cases, many participants renamed the test cases (Support: 7) and the used variable identifiers (Support: 3). For identifier names that should be renamed by the developer during the inspection, two participants promoted a clearer naming such as "`placeholderN`".

Code: Concise We observed that the code should be as **concise** as possible. Developers were confused by **unnecessary statements** (O#5, Support: 7), which were left over from the amplification process. Underneath them were object initializations or method calls no longer relevant for the intent of the new test case. The developers needed additional time to detect these statements as unimportant and to delete them (O#6, Support: 3).

We saw a similar effect **with unnecessary casts** (O#7, Support: 8) introduced by DSpot for type safety. In many cases, the casts could be identified as superfluous, but the developers were unhappy that extra work is necessary to remove obviously superfluous code.

For generated assertions that check the return value of a function, DSpot actively splits the function call and the assertion through a variable declaration. While one participant preferred this step for clarity and would find it easier to understand with an expressive variable name, three other participants were annoyed by the additional bloating of the code (O#8, Support: 3). While inlining the variable declaration a participant even pointed

out that splitting the function call from the assert statement lead to **less powerful assertions** being used (O#9). Instead of `assertNotEquals` the test used `assertFalse`, which gives a much less expressive error message in case the assertion fails. `assertNotEquals` throws a `org.junit.ComparisonFailure` and prints the expected and actual values, while `assertFalse` simply throws an `java.lang.AssertionError` at the line of the assertion.

Another issue brought up by the developers was **concise input strings**. To generate parameters while creating new objects for the setup phase of a test case, DSpot can generate unnecessarily long random strings of characters (O#10, Support: 2). To understand the test's behavior, developers now would need to know which of these characters are important for the intent of the test case, which takes a lot of time and effort. In one case we observed, DSpot created a new object and checked that it is not equal to the existing one. The developer spent a lot of time going through the various special characters in the constructor parameters for the new object, trying to determine which one is triggering the behavior of the test case. In the end, none of the special characters were necessary, the strings simply had to be different from the strings initializing the existing object. This anecdote points to the need for minimizing input values in generated test cases, which was also pointed out by Fraser et al. [11]

Code: Consistent Apart from the need for the test code to have a high quality [38] itself, it should also be **consistent** with the rest of the test suite. Three participants pointed out that the assertion methods were fully qualified instead of statically imported, like in the original test case (O#11, Support: 3). A similar comment was made for the identifiers, which one participant wanted to be in the same naming schema as existing test cases.

Behavior: Relevant Moving to the behavior of the test cases, we saw that it is important to test methods **relevant** to the developers. As our example project already had a relatively good test suite covering most core functions of the class, many of the proposed test cases covered extra branches in `equals`, `hashCode` or `clone`. The initial reaction of various participants was "I would not test this method" (O#12, Support: 11), leading some to discard the test directly. Others investigated further and uncovered that hashCode was overwritten with a custom implementation, which for one participant meant it was relevant after all to test the method. We believe that it is not only important to focus on testing methods important to the developer, such as core functions defined in a class, but also to make it clear why a newly covered function is considered important, e.g., because it overwrites the defaults with a custom implementation. How many interesting test cases are proposed was an important point for the developers we interviewed, this would majorly influence whether they keep using *TestCube* (O#13, Support: 4).

Behavior: Invariant A further comment on the behavior of the generated test cases was that developers would like them to test **invariants** of methods instead of absolute values (O#14, Support: 8). The most prominent example being `hashCode`. As other test generation tools, DSpot uses the current behavior of a system as an oracle. To test `hashCode`, it calls the method on an object and creates an assertion comparing the resulting value to the return value of `hashCode`. This is a fragile test case, as the `hashCode` changes as soon as any changes are made to the class's attributes. Our participants advocated testing the

invariant of `hashCode` instead of an absolute value. Interestingly though, they proposed a variety of ways how to test this invariant: Cloning an object and checking the `hashCode` is still the same (as well as that they are equals), creating the same object twice and compare the `hashCode`, check that if equals is true the `hashCode` is also the same, or even creating random objects and verifying that only a few of them lead to hashCode collisions. While proposing amplified test cases to open source projects, Danglot et al. also saw diverging reactions to test cases testing `hashCode` [13].

Behavior: Diverging One of our observations that is special to test amplification is how far the behavior of the generated test case should **diverge** from the original test case (O#15, Support: 6). Some of our participants were enthusiastic that the generated test cases explored so many new paths and scenarios, even naming this as one of the key strengths of *TestCube*. Others were confused by this divergence as they mainly focused on comparing the behavior of both test cases. The most severe cases of such a divergence approach when the original test case involves objects from another class than the class under test. Some of the amplified test cases then test functionality in this other class and completely disregard the original class under test (O#16, Support: 3). While these tests can be valid and helpful additions to the test suite, our participants mostly disliked them, because they were focussing on testing the original class under test. In a future version of *TestCube*, tests for another focal class should be marked as such and proposed to be added to the fitting test class.

RQ1: The Key Factors to Make Amplified Test Cases Suited for Developer-Centric Test Amplification

Summarizing the results and observations described in this section, the key factors to make amplified test cases suited for developer-centric test amplification are concerned with the code and the behavior of the test cases. When it comes to code, the variable identifiers and test names should be meaningful, the code should be short and concise, and consistent in terms of quality and style with the rest of the test suite. With respect to its behavior, an amplified test case should execute scenarios that are relevant for the developer, should test invariants in place of absolute values where possible, and should match the developer's expectation in terms of divergence from the original test case.

2.4.3 RQ2: What Are the Key Factors to Make Test Exploration Tools Suited for Developer-Centric Test Amplification?

To enable developers to interact with and judge the amplified test cases, we created a test exploration tool. In the following, we will explain our observations from the interviews on what factors are important for such a tool to be suited for developer-centric test amplification.

Ease of Use: Minimal Configuration First and foremost, a test exploration tool should be **easy to use** and especially **easy to start**. Our approach of using a **default configuration**

was successful, two of our participants pointed out how little effort was needed from their side to get started (O#17, Support: 2). Some still noted concerns about how easy the tool would be to set up locally for their projects (O#18, Support: 2), so clear supporting documentation is important if one wants to let the developer try out and discover a tool all by themselves.

Ease of Use: Integration A factor that helped developers start up and explore *TestCube* so quickly was its tight **integration** with IntelliJ (O#19, Support: 3). Participants noted that it was easy to start from the "run test" location, two made use of the built-in code navigation to explore the code under test (O#20, Support: 2) and one liked that they could perform all actions without having to switch tools (O#21, Support: 1).

Ease of Use: Usability In addition to minimal configuration and being integrated, a developer-centric exploration tool should also adhere to the long-established criteria for **usability** from Human-Computer-Interaction research [68]. We have seen that it is important to give the developer control over the layout of *TestCube*: various participants had different wishes for which information they want to see and how much space the tool should take up on their screen (O#22, Support: 1 each from 5 codes). Some were looking for buttons to close, e.g., the coverage editor they no longer needed (Support: 3) or got confused after they could not undo an unintentional action (Support: 3). We believe it is crucial for a successful tool to give the developer options to configure the layout of the tool to fit their needs and let them recover from errors.

With the help of the *System Usability Score*, we evaluated the overall usability of *TestCube*. 44 % of our participants rated the usability as "Excellent", 38 % as "Good" and 19 % as "Poor". This shows that even with the above mentioned issues, we overall succeeded in creating a tool that is easy to use.

Information Management We observed big difficulties with **managing the information** *TestCube* is displaying for developers. The text detailing which instructions are additionally covered was overlooked by many study participants, some later said they thought it is "unimportant debug output" (O#23, Support: 3). Providing the information sought out by developers in a way that does not overwhelm them and is accessible to them where they expect it is one of the big challenges looking at future versions of *TestCube*. Also the number of generated test cases should not be too large (O#24, Support: 4). For some methods, over fifty new test cases were proposed that one by one tested a previously uncovered class. One participant said they lost interest after looking over several of these test cases and seeing how many were left (O#25, Support: 1). An effective test exploration tool should focus on a few impacting test cases to not overwhelm the developer and keep each interaction session compact. Additionally it would help to rank the generated test cases and show the most impactful ones first.

Focus Related to information management is also the issue of **focus**. Through nearly each one of our interviews, we saw how important it is to only show information to the developer which they are supposed to focus on in that moment. In the current design of *TestCube*, the amplified test cases are all part of the same text file presented at once in

an editor. This means that multiple tests are visible at the same time. While *TestCube*'s internal navigation, e.g., the coverage information and the automatic adding to the test suite, was focused only on the first test case at the start, many of our participants started scrolling through the list of test cases immediately (O#26, Support: 5). Later some of them were confused (O#27, Support: 3), as they tried to add the test case they were currently focussing on to the test suite, while *TestCube* copied over the first one in the list. It is therefore extremely important for a future test exploration tool to make sure the focus of the developer aligns with the focus of the tool. For example, by only showing the code of one test at a time and therefore forcing the user to click on the next and previous buttons to explore the generated test cases.

Manage Expectations: Runtime A test exploration tool should **manage the expectations** of its users. We observed this with the **runtime** of the amplification process. Even though we took care in our configuration to keep the runtime of DSpot as low as possible, in some cases the generation still took several minutes to complete, which four participants considered as too long (O#28, Support: 4). While we included a business indicator that signaled to the developers that the amplification is running a background task, many were wondering how long it will take before they get results. Our participants wished for an expressive progress bar that either gives an estimation of the remaining time or at least shows an approximated form of progress (O#29, Support: 2). Some were wondering whether, or even expecting that, it is possible to switch to another task while they were waiting.

Manage Expectations: Capabilities The expectations with respect to the **capabilities** of a tool should also be correctly set. As we gave the developers only a minimal introduction to the tool, it was not clear for some whether the generated test cases are meant to replace the existing test case or are meant to be an addition to the test suite. Toward the end of their interviews, participants pointed out that they slowly understand the power of the tool better and see clearer how they would employ it (O#30). One pointed out that the generated test cases were much more appreciated by him now that he understood the editing effort which was necessary before including them.

Overall, we observed a plethora of important aspects to make a test exploration tool developer-centric. It should be easy to start and use, the way of displaying information needs to be carefully chosen, it has to keep track of the focus of the developer and manage the user's expectations towards its capabilities and runtime.

RQ2: The Key Factors to Make Test Exploration Tools Suited for Developer-Centric Test Amplification

In summary, a key factor to make test exploration tools suited for developer-centric test amplification is making the tool easy to use: through minimal configuration, through a tight integration into the developer's existing environment and through adhering to established usability principles. Such tools should manage the information they present to the developer and help the developer focus on the information they need for their current task. Further, a test exploration tool should manage the expectations their users have towards the runtime and the capabilities of the tool to ensure that these expectations can be fulfilled.

2.4.4 RQ3: What Information Do Developers Seek While Exploring Amplified Test Cases?

While exploring the generated test cases, our participants did not only scrutinize the test code itself but were also looking for and asking about a lot of additional information. We saw that it is crucial to provide quick and familiar ways for developers to provide this information so they can efficiently decide on whether to keep or how to adapt an amplified test case.

Test Case: Behavior / Intent As mentioned in Section 2.4.2, the test cases themselves and their **behavior** or rather their **intent** were a main focus of the developers. After making edits to a test case, one participant wondered whether the original intent of the generated case was still preserved (O#31, Support: 1). This is in line with Grano et al.'s results: developers are concerned with determining whether a unit test "actually exercises the corresponding unit" and how many relevant scenarios are covered [61]. Prado and Vincenzi showed that the code of a test case is one of the main sources of information about a test case for the developer [69], an observation corroborated by Aniche et al. [28]. As far as we observed, the current editor displaying the code of the generated test case is enough to satisfy this information need for developers.

Test Case: Outcome Furthermore, the developers were interested in the **outcome** generated test cases (O#32, Support: 3), i.e., whether they are passing or failing. As all test cases generated by DSpot pass, this could be addressed by a better explanation of the tool. Alternatively, tool developers could provide the existing IDE utility to run a test case in the editor proposing the new test case or provide functionality such as Infinitest, a tool that runs JUnit tests continuously in the background [70]. This would allow the developer to easily check that a test case is still passing after editing it and before integrating it to the test suite.

Test Case: Runtime The **runtime** of a test case was also pointed out by one of our participants (O#33, Support: 1), as they were used to projects where increasing the runtime of the continuous integration build was frowned upon. Test exploration tools should include a note about the measured execution time with each test case.

Code Under Test While inspecting the new test cases, most of our participants quickly jumped to also inspecting the **code under test**. Two were trying to understand its behavior (O#34, Support: 2) to see the intent of the test case and to judge whether the additional coverage was relevant. Also, they checked if the tested method overrides standard behavior and if exceptions were thrown and tested. As Spadini et al. already pointed out, it is crucial for test review tools to provide easy navigation between test code and the code under test [59]. Prado and Vincenzi point out that developers should receive tool support to build the context between test code and code under test [69]. Through reusing the standard editor component of IntelliJ, *TestCube* allows its users to use their familiar code navigation tools, such as command-click to go to the definition of a method.

Coverage The third large area developers wondered about was **coverage**. Under this falls the original coverage of the test suite and which additional coverage each generated test case and all the generated test cases together yield. A recurring question was whether a functionality covered by the generated test cases was already covered by another test case (O#35, Support: 2). Developers scanned the test suite to find other tests calling the same method. Even though *TestCube* provides detailed information on which instructions are additionally covered by the new test case, not all participants understood that this implies the instructions were not covered by any existing test case. The developer that found our visualization of the added coverage, found it helpful to see the covered lines not only as numbers but also in the code context (O#36, Support: 1). They wished for a separate report of the original instruction coverage and the improvement of instruction coverage after including the amplified test cases. We also observed two times that the developers used the coverage of a test case to infer its intent (O#37, Support: 2), an observation also made by Grano et al. [61]. Sometimes it was unclear to our participants why the new code covers these additional instructions (O#38, Support: 3). This points towards a need for exploration tools to visualize clearer how test code and code under test connect.

Instruction coverage seemed to be a satisfying metric for most of our participants. Some of them wished for more information about how many branches are covered or hoped the tool would help them cover all branches as they would aim for while writing unit tests themselves. Even though some of our participants were aware of the concept of mutation score, none of them asked for information about improved mutation score or for test cases that kill additional mutants. Rojas et al. saw that in an industrial context many developers used coverage to evaluate a generated test case [47].

Original Test Case Special for the case of test amplification, was the interest of the participants to inspect the **original test case** that was the basis for the amplification. They tried to understand the intent of the original test case (O#39, Support: 5) and used this information, together with the knowledge of which instructions were changed to determine the behavior of the amplified test cases (O#40, Support: 2). Highlighting the changes from the original to the generated test case through comments (C#4) was not successful in our study. The developers ignored them and questioned their usefulness (O#41). We hypothesize that the generated test cases were short enough to spot the changes without the comments.

RQ3: The Information Developers Seek While Exploring Amplified Test Cases

In our interviews we observed that developers are interested in a wide array of information while exploring and inspecting amplified test cases. For a test case itself, developers try to understand its behavior and intent, ask whether it is passing and how long it takes to execute. Beyond the test case, they are concerned with the code under test, the original and added coverage as well as the original test case the amplification was based on.

2.4.5 RQ4: What Value Does Developer-Centric Test Amplification Bring to Developers?

One way to make developer-centric test amplification successful, is to bring across the value they can expect from the amplified test cases and from using the test amplification tool. To give us an indication, which values we should focus on, we collected comments from our participants about the benefits they believe they would achieve from using a tool similar to *TestCube*.

Improve Test Suite: Ease Test Engineering First and foremost, automatic test amplification would help them **improve their test suite**. By proposing complete, ready-to-run test cases that cover more code or interesting behavior (O#42, Support: 4), automatic test amplification **eases test engineering** for developers. The test amplification alleviates the developer from having to write test cases from scratch, reducing the effort necessary to develop a test suite. Reducing effort is a concern for developers: one participant stated, that they would "either look for less work or for tests with a better quality" (O#46, Support: 1).

Improve Test Suite: Inspiration The generated test cases also provided **inspiration**. Several users created new test cases to cover the behavior of the amplified test cases (O#47, Support: 4). They were glad to be pointed to untested code paths (O#43, Support: 4) and to unexpected scenarios that could happen in the system (O#44, Support: 1). A recurring comment was that a test covers methods the participant always forgets to test (O#45, Support: 3). By proposing new scenarios with the generated test cases, test amplification tools can take the burden of designing test scenarios of the developers.

Learning Packaging test case generation in an easily accessible plugin can be a valuable step to enable more developers to **learn** about test amplification itself. Many of our participants did not know about the technique of test amplification before and one said that a plugin like *TestCube* could be a way to bring this idea into industry (O#48). The participants got more confident towards the end of the interviews about what *TestCube* can do for them and how they could apply it effectively (O#30). In general, we saw that amplification was easy to grasp for the developers (O#49). A participant pointed out they would like to use such a tool while they are working on improving the test suite (O#50, Support: 1) and another was eager to try it on their own projects (O#51, Support: 1).

Confidence One participant said that using *TestCube* more often would increase their **confidence** in their test suite (O#52, Support: 1). On the one hand simply through the higher coverage after adding the generated test cases, and on the other hand because they see more important scenarios being covered.

RQ4: The Value Developer-Centric Test Amplification Brings to Developers

Our participants named a number of benefits they would gain from using an automatic test amplification tool regularly. It would make it easier for them to develop test cases, by alleviating them from the effort to write the test cases and by providing inspiration of scenarios they tend to forget to test. A developer-centric test amplification approach would support them learning about automatic test amplification and using it would increase their confidence in their test suite.

2.5 Discussion and Recommendations

In the following, we consolidate our results into two actionable recommendations on how to make amplified test cases and test exploration tools suited for developer-centric test amplification. Table 2.3 shows from which of our interview observations we infer the recommendations.

We chose an additional layer between the test amplification and the developer, a test exploration tool, to address the issues users previously reported with DSpot. Our prototypes could already surface different kinds of information the developers were seeking, such as the behavior of the test case (O#2), its coverage (O#35), or which code is tested (O#34). Further, it became clear how tightly the characteristics of the test exploration tool are bound to the kind of test cases it presents to the developer. In our design, the technique of test amplification (C#1) and the information from the amplification process reports (C#9), is tightly bound to what our exploration tool *TestCube* presents to its users about the test cases (C#14). Our participants were questioning how the test cases are generated, and also sought information especially related to test amplification, like the original test case (O#39) (O#40). We saw the importance of expectation management (O#30), e.g., on how much they should edit the proposed test cases, and conveying the value the test amplification can bring to the developer, such as pointing to untested code paths (O#43). The tight integration into the developer's IDE was helpful to get them started quickly (O#19). Overall, we saw a positive effect of using a test exploration tool to facilitate the developer-centric test amplification. We conjecture that this support of an integrated test exploration tool is also beneficial for other test generation approaches that aim to be developer-centric. We recommend to future authors of developer-centric test generation approaches to provide a test exploration tool that is targeted towards the test generation method they employ and accessible to the developer from their familiar environment.

Recommendation	Corresponding Observations
Recommendation 1: Consider the interaction of the developer with the test cases: provide a test exploration tool that is targeted towards the test generation method and integrated into the developer's environment.	(O#2) (O#35) (O#34) (O#39) (O#40) (O#30) (O#43) (O#31) (O#24) (O#26) (O#27) (O#28)
Recommendation 2: When the main goal is for developers to accept a test case into their maintained test suite, it is more important that the test case is understandable and relevant to the developer, than how much it impacts the coverage of the test suite.	(O#2) (O#3) (O#5) (O#7) (O#12) (O#35) (O#14) (O#11) (O#9) (O#10)

Table 2.3: Connection of our recommendations to our interview observations.

Recommendation 1

Consider the interaction of the developer with the test cases: provide a test exploration tool that is targeted towards the test generation method and integrated into the developer's environment.

Concretely, *TestCube* can be improved in several points: Clearer visualization of the connection from the amplified test case to the additionally covered instructions in the code under test (O#23) (O#34) (O#38) and describing the behavior of the amplified test case and how it diverges from the original test case (O#31). Further we can help developer focus by proposing one test case at a time (O#24) (O#26) (O#27) and address waiting time (O#28) by generating test cases before they are requested.

Looking at the results of our first research question, we can see that the developers were mainly concerned with *understanding* the test cases *TestCube* presented to them (O#2). The observations which occurred in most interviews are about the identifiers (O#3), the conciseness of the code (O#5) (O#7), and trying to understand the behavior and intent of the test case (O#31), be it through the test code itself or the various other kinds of information sought. During our interviews, the understanding was always the first step—only after the participants understood a test case they started to judge the impact or relevance (O#12) (O#35). When judging the test cases, we observed that not all tests which increase instruction coverage are relevant to developers, e.g., because they test a to them less important method (O#12) or a too narrow behavior (O#14). From these results, we infer that for a developer-centric approach, where the central aim is for a developer to take over the generated test case into their maintained test suite, the understandability of the generated test case and the relevance to the developer is of a bigger concern than how high its numeric impact is on the coverage of the test suite. An understandable test case with a weaker coverage contribution is more likely to be accepted by developers, compared to a test case that increases coverage greatly but they discard because they can

not understand what it does. We recommend to future authors of developer-centric test generation approaches to prioritize the understandability of the generated test cases and their relevance to the developer higher than their impact on the coverage of the test suite.

Recommendation 2

When the main goal is for developers to accept a test case into their maintained test suite, it is more important that the test case is understandable and relevant to the developer, than how much it impacts the coverage of the test suite.

Concretely, the amplified test cases generated by DSpot can be improved by generating useful identifiers (O#3), possibly informing about the unique coverage provided by the test case (O#35) [33]. Further unnecessary statements (O#5) and casts (O#7) should be removed [71], the style of the test cases (O#11) can be adapted to fit the existing test suite, randomly generated strings shortened and focused to the part triggering the tested behavior (O#10), and assertions adapted to use the most specific assertion giving an informative error message (O#9).

Our participants pointed out how easy it was to interact with *TestCube* right from their IDE (O#17) (O#19) (O#20), many found test cases that they liked and added them into the test suite of our example project. We conjecture that combining the already powerful state-of-the-art test amplification approaches with well-designed, developer-centric test exploration tools will let us reach more developers to amplify their software testing practice.

2.6 Threats to Validity

There are several threats to the validity of our results which we discuss in this section.

Confirmability To ensure that our results are formed by the interviewees and not by the authors, we base our results as closely as possible on the interviews. While coding and analyzing the interviews we performed extra steps to validate the codes elicited from the interviews and evaluated the inter-rater reliability, as described in Section 2.3.3. Nevertheless, other researchers might structure the resulting codes differently or draw varying conclusions from them. We publish the full codes together with their frequency in our interviews [64] for others to further explore the research area and add to our study.

Reactivity and Respondent Bias As the first author created the prototypes and conducted all interviews, the statements of the participants might be influenced by the participants wanting to please the creator of the tool they are evaluating. To mitigate this threat, we repeatedly invited the participants to be critical and refrained from defending the current state of the tool. Based on the wide variety of critical and positive points we could collect, we conjecture to have mitigated this threat.

Construct Validity A threat to the construct validity of our study is that our participants interacted with an early prototype showing one possible design of a test amplification tool. Bugs in the prototype or design decisions we took could influence the developer's experience and the generalizability of the results to general test amplification approaches. We identify

several of our results as being related to our choice of test amplification to generate the test cases (C#1), which we indicated while reporting them in Section 2.4. Similarly, our observations can be influenced by our default configuration of DSpot. Optimizing the configuration of DSpot to fit the target project would likely lead to more relevant test methods being generated. Furthermore, our participants were not developers of the example project we used in the study. We expect that developers familiar with a project would spend less time on understanding the original and amplified test cases and could judge more easily if a production method is relevant to be tested.

Dependability Whether our results are consistent and can be repeated in a replication is the concern of dependability. With 16 participants we were able to interview a relatively large number of software developers. Our presented results mainly focus on observations that we made in multiple interviews (that have high support). Nevertheless, there were many insightful comments that only emerged from one or a few interviews. Through the openness of our setup and questions, the interviews went in many different directions and the observations we could make are dependent on the taken direction. We expect that repeating this study would yield different support for the rarely-mentioned aspects, however the overall conclusions will likely stay the same.

External validity There are several threats to the generalizability of our results. As well as other state-of-the-art test generation tools, our prototypes address Java and its specific properties. We expect our results to generalize to other object-oriented, statically typed languages and are curious to see the different information needs developers of other programming languages have.

The choice of presenting our prototypes together with the example project Jsoup can also impact our results. Because equals, hashCode and clone were not covered by the existing test suite, DSpot generated tests mainly for these functions that were named "irrelevant" by several of our participants. In other projects whose test suite has a lower or differently distributed coverage, the aspect of testing relevant methods might be less apparent.

As we performed convenience sampling, the results of our study might be influenced by our professional networks, as well as a self-selection bias of developers that are especially interested in high-quality test suites. From the demographic information we collected, we conclude that we sampled from a broad variety of experiences, industry domains and team sizes.

2.7 Related Work

Various past works have focused on the two main parts of or approach, mainly improving the understandability of test cases and integrating test generation tools into development environments.

2.7.1 Understandability of Test Cases

The issues of cryptic identifiers and lack of documentation in generated test cases are addressed by Roy et al. [34] in their tool *DeepTC-Enhancer*. With a combination of templates

and deep learning, they generate comments that explain the behavior of a test case and meaningful identifiers. Their work is an extension of *TestDescriber* by Panichella et al. [35] and was evaluated by 36 developers. The developers were most enthusiastic about the meaningful identifiers, while some said the explanatory comments are not concise enough. In our interviews, we also observed the importance of expressive test names and variable identifiers. While our participants were trying to understand the behavior of the amplified test cases, they could interpret the raw code of the test cases well. Therefore, we do not believe any additional summarization of the test itself is necessary. Easier access to the code under test and information about previous and added coverage are more relevant concerns going forward. In similar vein, Li et al. describe UnitTestScribe [72].

Alsharif et al. [73] investigated which factors are important for the understandability of automatically generated SQL schema tests. They saw that human-readable string values are better to understand than randomly generated ones and the repetition between generated test cases made it easier to focus on the relevant differences of the test cases towards each other. Their results align with ours: Randomly generated strings were mentioned as confusing and our interview participants repeatedly used the similarity between the original and the amplified test case to understand the behavior and impact of the newly generated test case.

Daka et al. [29] define a regression model for test case readability based on various syntactic properties of test cases. They integrate the model into the fitness function of EvoSuite [10] to generate more readable test cases. In their model and post-experiment survey they identified several important factors overlapping with our findings: Identifiers are important for the understandability of a test case, as well as no unnecessarily defined variables and short string literals.

Next to *DeepTC-Enhancer* by Roy et al. [34], several further works focus on generating meaningful names for test cases. *NameAssist* by Zhang et al. [31] infers test names from the class under test, the expected outcome stated in the assertion and the overall test scenario defined in the body of the test. Daka et al. [32] derive test names from additionally covered exceptions, methods, outputs and inputs of the component under test. They showed that the generated names are equally excepted compared to names given by developers and made it easier for developers to match a test to the code under test. Including an advanced name generation approach such as the one by Daka et al. [32] would be a valuable addition to *TestCube* and DSpot.

Bihel and Baudry [36] focused specifically on making tests amplified by DSpot more accessible for developers. They generate a natural language description of the changes made during the amplification, of the value observations which lead to new assertions, and of the mutants which will be killed by the newly added test cases. These descriptions are designed to accommodate a pull request proposing to add an amplified test case. In comparison, *TestCube* focuses on a just-in-time interaction of the developer, embedding test amplification into their IDE. In our scenarios, not only tool performance, but also the amount of presented information is a distinguishing challenge. Compared to Bihel and Baudry's approach [36], *TestCube* more carefully selects the information presented to the developer. We also evaluate our approach in a study with developers.

Because of the high computational cost of test generation, many tools have opted for integration into the continuous integration process [18, 74]. This, however, leads to a

long time distance between triggering the test generation and receiving results [75], as well as the developers having to inspect the tools outside of their familiar development environment. To provide more immediate value and direct feedback, we opted to let *TestCube* run on our user's computers, leading to many more constraints regarding the available execution power and therefore possible complexity of the applied algorithms.

2.7.2 Test Generation Tools Integrated in the IDE

Several other test generation tools have been integrated into IDEs up until now. Following an industrial study of EvoSuite, Rojas et al. [47] pointed out the importance to integrate test generation tools into development environments. Since then, EvoSuite has been lightly integrated into IntelliJ IDEA as a plugin [74] which provides options to configure the test generation within an existing build process. In contrast to this, *TestCube* runs independently from a project's build process and can be installed and applied with nearly no configuration[10].

DSpot has been integrated into the Eclipse IDE as a plugin together with other tools from the STAMP project [76]. The plugin offers a graphical interface to set the various configuration parameters of DSpot and start the amplification process. Compared to *TestCube*, the additional information showing the impact of a generated test case is just presented as a JSON text and the developer is still confronted with many configuration parameters.

Tillmann and de Halleux [12] developed the Pex tool which generates inputs for parameterized tests based on program analysis. They integrated their tool into Visual Studio, enabling the developer to generate and execute the unit tests by right-clicking on the parameterized unit test. The tool presents the generated inputs and corresponding test results in a new window as a simple table.

2.7.3 Interactive Test Generation

The idea of connecting the developer closer with the test generation is also realized in *Interactive Search-Based Software Testing (ISBST)*. In the concept of Marculescu et al. [77], domain experts decide the importance of different components in the fitness function leading the automatic optimization of the test cases. The interaction happens during the search process, where in defined moments the expert evaluates the current candidate test cases and adapts the fitness function for the next round of test generation. When compared to manual testing, ISBST could find different test cases and execute behavior previously not considered by developers, similar to what our participants reported about *TestCube* (O#45). Marculescu et al. also investigated the mental workload of developers using ISBST compared to manually writing test cases. They did see a higher load and explain it through the distance from the developer's interaction with the fitness function to the outcome of the search process. Similarly, we saw during our interviews that the developers tried to retrace the generation of the test cases, strengthening the choice to perform only small edits during the amplification to make the process easier to retrace. After transferring their approach to industry [78], Marculescu et al. point out the need for ISBST and other automated test systems to effectively communicate their results to their users. Our work

[10] In the current version the user only has to provide the path to their Java 8 installation and their Maven Home.

addresses this by prototyping a developer-centric test exploration tool and eliciting the key factors to make such tools suited to be used for test amplification.

2.8 Conclusion and Future Work

With this chapter, we are setting a step towards test amplification that is centered around the developer and their needs. Based on reported issues with current state-of-the-art tools, we devised design intentions for a developer-centric test amplification approach that aims to generate test cases that will be taken over into the manually maintained test suite. We used these intentions to adapt DSpot's test amplification and create *TestCube*, a powerful test exploration plugin for IntelliJ. With the help of these tools, we interviewed 16 software developers from a variety of backgrounds and collected detailed insights on how the amplified test cases and the exploration tool should be adapted to best fit their needs. Through evaluating the information sought during the test exploration, as well as the value test amplification brings to developers, we guide future tool developers on what they should bring forward in their upcoming, developer-centric test generation tools. We summarized our observations and results into two recommendations: Tool makers should consider the interaction of the developers with the amplified test cases and provide a targeted and integrated test exploration tool. If taking over the test cases into the maintained test suite is the declared goal, the understandability of the amplified test cases should be prioritized over optimizing the coverage of the test suite.

In short, we contribute:

- two recommendations on how to design developer-centric test amplification tools
- a structured overview of the key factors to make amplified tests as well as test exploration tools suited for developer-centric test amplification
- a refined, developer-centric test amplification approach, based on the DSpot test amplification
- a developer-oriented test exploration plugin for the IntelliJ IDE

Going forward we want to understand the different aspects of developer-centric test amplification in more depth. We want to look into generating meaningful identifiers fast enough, ranking test cases according to their relevance to the developer, and providing them information such as runtime or coverage when they look for it, but without overwhelming them. Our tools will dive deeper into their day-to-day development, for example by helping them incrementally generate test cases for new or untested classes. We want to give them more power to direct the amplification and receive test cases that cover code or scenarios they are interested in, while also providing them with subtle, helpful recommendations before they realize they need another test case. Our vision is to build tools and methods that empower developers to create better test suites with less effort, while they are at the steering wheel deciding over, leading, and benefiting from our automatic test amplification.

Acknowledgements We would like to thank the participants of our study for the valuable feedback on our work. This work was sponsored by the Dutch science foundation NWO through the Vici "TestShift" project (No. VI.C.182.032).

3

How Does This New Developer Test Fit In? A Visualization to Understand Amplified Test Cases

Developer testing, the practice of software engineers programmatically checking that their own components behave as they expect, has become the norm in today's software projects. With the constantly growing size and complexity of software projects and with the rise of automated test generation tools, understanding *a test case is becoming more and more important compared to writing test cases from scratch.*

This holds especially in the area of developer-centric test amplification, where a tool automatically generates new test cases to improve a developer-maintained test suite. To investigate how visualization can help developers understand and judge test cases, we present the TestImpactGraph*, a visualization of the call tree and coverage impact of a JUnit test case proposed for amplification. It empowers the developer to drill down into the behavior of a test case, as well as providing them a clear view on how the proposed test case contributes to the coverage of the overall test suite. In a think-aloud study we investigate which information developers seek from the* TestImpactGraph*, how its features can support them in accessing this information, and observations regarding the coverage impact of test cases. We infer ten actionable recommendations on how developer tests can be visualized to help developers understand their behavior and impact.*

This chapter is based on *C. Brandt and A. Zaidman. How Does This New Developer Test Fit In? A Visualization to Understand Amplified Test Cases, Working Conference on Software Visualization (VISSOFT), 2022* [53].

Developer tests—xUnit test programs which developers use to check the behavior of their code [3]—have become a cornerstone in assuring the quality of today's software systems [8]. As test suites are growing in number and size, understanding test cases one has not written themselves is becoming more and more important, for example, (a) when trying to understand a failing test [35], (b) when using developer tests as a form of executable documentation [39, 40, 58], (c) when test cases are submitted for code review [59], or (d) when determining whether to add an automatically generated test case to the test suite, e.g., checking whether the captured behavior is correct [11, 45], [Chapter 2].

Point (d) is especially important in the area of *developer-centric test amplification* [Chapter 2]. Test amplification is the process of improving an existing test suite with the help of automated tooling [42], in our case automatically generating new test cases that strengthen a manually written test suite. In *developer-centric* test amplification the goal is to partially relieve the developer's effort in writing test cases by generating ones that the developer subsequently takes over into their maintained test suite [Chapter 2]. In this process, it is important that the developer understands the new test cases, and subsequently accepts or rejects them based on their understanding of the added value. We want to illustrate this with an example:

> Sara is a software developer who wants to improve her test suite with the help of an automated tool. The tool that she uses generates a few new test cases that supposedly improve the coverage of her test suite. Next, Sara browses through these test cases to determine whether they make sense and test behavior that is correct and relevant for the software under test. She does not only want to understand what the test case does, but also how it improves her current test suite. Even though the tool tells her in which lines new instructions are covered, she has to click and search through the called methods one by one to understand how the test case reaches these new instructions. Sara wishes that there was an easier, less time-intense way to understand the test cases.

To help developers such as Sara explore, understand, and judge test cases that amplify an existing test suite, this chapter presents the TestImpactGraph. It enables developers to drill down into the methods called by a test case without having to jump from file to file and risk loosing their mental context. A clear indication of where the test case contributes additional code coverage, helps the user judge whether including the test case improves their test suite.

With the help of the TestImpactGraph we conduct a think-aloud [79] study to investigate what software developers expect from such a visualization of developer tests. In this chapter, we present our results, focusing on the information developers seek from the TestImpactGraph, features that help developers access this information and observations related to test coverage that arise from inspecting a test case through the TestImpactGraph. We discuss how the TestImpactGraph could be applied in further scenarios, like inspecting a proposed test from a pull request, and give ten actionable recommendations on how tools should visualize the behavior and impact of developer tests to aid the software developers exploring and understanding them.

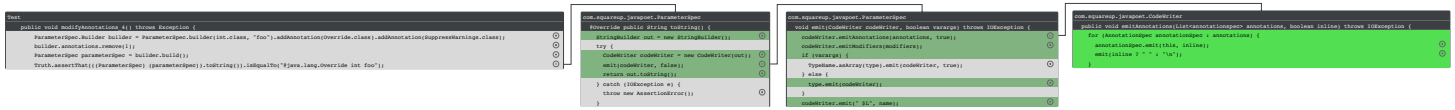


Figure 3.1: An example TestImpactGraph, visualizing the second test case in our study.

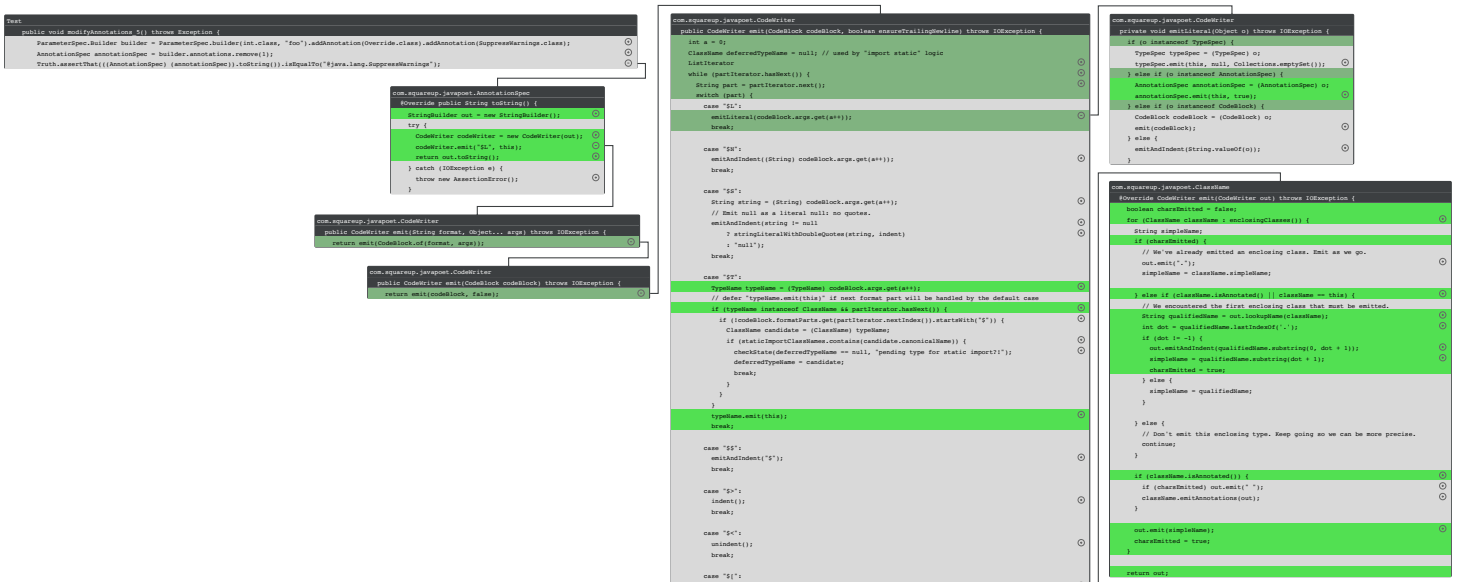


Figure 3.2: The scattered coverage highlights of the TestImpactGraph for the third test case in our study. To show more within the figure, we rearranged the first four nodes and cropped the fifth node.

3.1 Developer-Centric Test Amplification

This work builds upon or idea of *developer-centric test amplification* which we introduced in Chapter 2. To generate test cases that are accepted by developers into their manually maintained test suite, we adapted Danglot et. al.'s [13] test amplification approach to produce simple, focused new test cases that improve the instruction coverage of a test suite. In addition, we prototyped a *test exploration tool* which facilitates the interaction between the developer and the automatic generation tool. The process of using a test exploration tool is illustrated in Figure 1.3. We conducted semi-structured interviews to uncover what factors are important for their approach and the test exploration tool to be successful. They especially focused which information the developers looked for when judging whether it is worth to accept a test case.

Two key concerns for the study participants were the behavior and intent of the test case: What does it (aim to) test? Likewise, they wanted to know the test case's impact on the coverage of the test suite. During the interviews, the authors observed that the participants used the newly covered lines to infer the intent of the test case: The instructions that only the new test case covers must be what the test case is testing. In some cases, the methods with new coverage were not called directly by the test case, but only indirectly through changes in the input to other methods. The developers struggled to connect these test cases to the coverage impact they provided.

To augment this central interaction in developer-centric test amplification, we aim to develop a visualization that supports developers when inspecting the behavior and coverage of an amplified test case. The goal of this visualization is to:

- connect a test case to the methods it is executing,
- present which parts of the code are covered only by this test case, and with that
- effectively let the developer understand the behavior, intent and coverage of the test case.

Overall, the visualization could be part of our proposed test exploration tool, serving as one of several components that help the developers browse and judge amplified test cases to decide which ones to take over into their test suite.

3.2 The Test Impact Graph

In this section, we present the design of the TestImpactGraph, a visualization that supports developers in understanding the behavior and the coverage impact of a developer test. The graph consists of *nodes*, which represent the test case and the methods under test, as well as *edges* which represent method calls. The *default layout* helps developers directly focus on the additional coverage a test provides, while the *interactivity* lets them explore the behavior of the test case. Figure 3.1 shows an example of a TestImpactGraph.

3.2.1 Method Nodes

Each method, including the developer test which is visualized by the TestImpactGraph, is presented as a node. A node consists of the fully qualified class name, the signature of the

method and its source code. Figure 3.3 shows an example of a node presenting a method under test. The background of each source code line is colored depending on its coverage:

- **grey:** Not covered by this test case or belongs to the test code.
- **dark green:** Covered by this test case and already by another test case.
- **bright green:** Contains instructions only covered by this test case, which we call *additional coverage*.

With the term **additional coverage** we refer to those code elements that are covered by the inspected test case, but not by the other tests in the test suite.

```
com.squareup.javapoet.CodeWriter
  private void emitLiteral(Object o) throws IOException {
    if (o instanceof TypeSpec) {
      TypeSpec typeSpec = (TypeSpec) o;
      typeSpec.emit(this, null, Collections.emptySet());
    } else if (o instanceof AnnotationSpec) {
      AnnotationSpec annotationSpec = (AnnotationSpec) o;
      annotationSpec.emit(this, true);
    } else if (o instanceof CodeBlock) {
      CodeBlock codeBlock = (CodeBlock) o;
      emit(codeBlock);
    } else {
      emitAndIndent(String.valueOf(o));
    }
```

Figure 3.3: A node in the TestImpactGraph.

3.2.2 Call Edges

For each line with one or more method calls, the TestImpactGraph shows a small plus marker at the end. When the user clicks on this marker, they expand the edges connected to that line of code, showing all method nodes called by that line. The marker transforms into a minus icon, which lets the user collapse this part of the call tree again.

Apart from opening and closing call edges, the user can freely rearrange nodes, as well as drag and zoom the canvas to explore the TestImpactGraph. Initially, the nodes are presented in a hierarchical tree layout from left to right, with the inspected developer test as the root on the left.

3.2.3 Default Layout

As it is common for developer tests to execute a not-small number of methods [43, 44], the visualization we describe up until now can get quite large—and therefore overwhelming for the user. To clarify how the inspected test case improves the existing test suite, we

want to let the developer focus on what distinguishes the test case from the rest of the test suite. We use *additional coverage* for this. By default, the TestImpactGraph shows all methods under test that contain instructions that are covered by the inspected test case, but are not covered by the rest of the test suite. To give context on how these methods are called by the developer test, we also show all the method nodes on the call chains from the developer test to the methods with additional coverage. Figure 3.1 shows and example of this: the methods leading to the additional coverage are visible, while all other edges are collapsed.

3.2.4 Design Rationale

The design of the TestImpactGraph is based on various well-established visualization metaphors. To ease the adoption by software developers, we apply as many familiar visual components as possible and strive for a simple design that can fit within an IDE environment.

From related works, we know that developers who inspect a test case during code review are interested in the code under test [59]. They rely on source code to understand the system under test [28, 80] and should be supported while building up the mental context between test code and code under test [69]. This is why we present the developer test alongside the code under test.

We choose to directly show source code to the user, as this provides the highest code proximity [81]. In our interviews described in Chapter 2, we observed the developers navigating through the code using "jump-to-definition", a common strategy during code comprehension [82–85]. To let the developers keep the mental context of the methods they viewed, we show methods as rectangular nodes on a plane and use arrows to show calling relationships, similar to the Code Bubbles metaphor [86]. Just as Bragdon et al. observed with Code Bubbles, we want to support developers in *"understanding a call graph encompassing a handful of functions"* [87].

To visualize code coverage in an intuitively understandable way, we use the established notion of lines highlighted in green [88]. As many developer tests execute several methods [43, 44], presenting this large amount of information at once could be overwhelming for the user [81]. This is why we provide a default view focused on the most relevant methods—the ones with new coverage—and include interactive features [80, 89], letting the user zoom and pan to get an overview or a detailed look on items of interest. Markers to open and close branches indicate options for further exploration [81, 90], enabling the developer to access more details on demand or to filter out uninteresting elements.

3.2.5 Implementation

We implemented the TestImpactGraph as an extension to the TestCube plugin[1], which generates JUnit tests with the help of the test amplification tool DSpot[2]. We collect the method calls to build the TestImpactGraph based on a static analysis, using the IntelliJ PSI support[3]. The coverage information is provided by Jacoco[4] and obtained by DSpot

[1] https://github.com/TestShiftProject/test-cube/tree/v1.0.3-tig.1
[2] https://github.com/STAMP-project/dspot
[3] https://plugins.jetbrains.com/docs/intellij/psi.html
[4] https://www.jacoco.org/

during the test generation. The visualization itself is implemented with the G6 framework by antv[5] and can be found on GitHub[6] together with the exemplary graphs we used for our evaluation.

3.3 Think-Aloud Study

To understand the current state of the TestImpactGraph and collect feedback on what to improve and develop further to effectively help developers explore and understand test cases, we perform a preliminary think-aloud study. We aim to answer which information developers seek while they explore a proposed test case and judge whether it improves their current test suite (**RQ1**). Further, we want to know which existing and potential future features of the TestImpactGraph help developers effectively and efficiently access the information they are seeking (**RQ2**). Finally, we conjecture that the novel view TestImpactGraph provides at developer tests, will raise observations reflecting on the (additional) coverage of test cases (**RQ3**).

In summary, our preliminary think-aloud study intends to answer the following research questions:

RQ1: Which information do developers seek from the TestImpactGraph?

RQ2: Which features of the TestImpactGraph help developers access this information?

RQ3: What observations related to test coverage arise when inspecting a developer test through the TestImpactGraph?

3.3.1 Study Design

In our think-aloud study, we invite participants familiar with Java to inspect example test cases with the TestImpactGraph.

During the study, we go through three example test cases with each participant. The examples are amplified test cases which were generated by the test amplification plugin TestCube[7] to improve the test suite of the project `javapoet`[8]. We select this project for our study as we expect our participants to be familiar with its domain: generating Java source files. We aim to broadly explore the capabilities of the TestImpactGraph and therefore select three proposed test cases that show different patterns in terms of their additional coverage:

- The first proposed test case covers additional instructions in several lines, but not all lines, of a method that is directly called from the developer test. Its TestImpactGraph is shown in Figure 3.4.

[5] https://g6.antv.vision/en
[6] https://github.com/TestShiftProject/test-impact-graph/tree/v0.1.0
[7] https://github.com/TestShiftProject/test-cube
[8] https://github.com/square/javapoet

- The second proposed test case (Figure 3.1) covers additional lines in a method that is three calls away from the developer test.
- The third proposed test case has the most complex additional coverage pattern. It covers additional instructions in a directly called method, but also several method calls further away and on more than one branch of the TestImpactGraph. The TestImpactGraph in Figure 3.2 shows how scattered the additional coverage of our third test case is.

All three of these example test cases can be found on GitHub[9], their TestImpactGraphs can be explored as part of our replication package[10].

Generating meaningful names for automatically generated test cases and the variables used in them is a challenging and actively researched topic [31, 32, 34]. As test names and variable identifiers play a big role in understanding code [91], we do not want our study to be influenced by the quality of the automatically generated names. Therefore, we choose to simplify the test names to the name of the original test case and a number and to simplify the variable names to lowercase variants of their class.

Figure 3.4: The TestImpactGraph for the first test case in our study.

3.3.2 Study Execution

We used convenience sampling to recruit our participants: Four PhD Students from the field of computer science and one industrial software developer, all with two to five years of experience in software development and testing. They were unfamiliar with the example project `javapoet`. Before each session, we asked the participant for informed consent according to the ethics guidelines of our university.

Then, we gave a short introduction about the aim and the features of the TestImpactGraph. The TestImpactGraph is built to help developers understand an amplified test case, and especially the value it adds to the existing test suite, similarly to Sara in Chapter 3. This is why we ask the participants to answer the following task for each test case: *"What scenario is newly covered by this test case?"*.

While they browse through the visualization, we ask them to think aloud about their expectations and experiences. They stopped thinking aloud often during the experiment, as they sunk into understanding the code in front of them. After stimulating them with questions, they provided us with rich insights about the TestImpactGraph.

After the sessions, we analyzed the observer's notes using open and axial coding [66]. The results presented in Sections 3.4, 3.5 and 3.6 are grouped along the resulting axial

[9] L.68, L.92, and L.199 in `https://github.com/lacinoire/javapoet/blob/2cbb3084c15a209d28fc8c5fd7472dd695c22591/src/test/java/com/squareup/javapoet/generated/ParameterSpecTest.java`

[10] https://doi.org/10.5281/zenodo.6644723

codes. All codes and groups, as well as which participant mentioned them, can be found in our replication package.

3.3.3 General Observations

The participants liked interacting with the TestImpactGraph and appreciated a tool that lets them dig deep into the behavior of one test case. Overall, most of them intuitively understood the different components of our visualization: method nodes, call edges, the coverage highlights, and the default view of all additionally covered lines. One participant reported that such a visualization would let them be more confident in automatically generated code because they could retrace its behavior.

3.4 RQ1: Which Information Do Developers Seek From the TestImpactGraph?

In this section, we discuss our observations related to the kinds of information our participants sought while inspecting the developer tests to answer which additional scenario is covered by them.

3.4.1 What Does It Do? Understanding the Test Case

From observing our participants, we learned that different developers focus on different parts of a test case or its execution when they explore the test case and determine its behavior or its impact:

- **Test names:** Several participants based their judgment on the names of the test class or the test case. As we simplified the variable names (see Section 3.3.1) these could not be used as a source of additional information.
- **Test code:** Several times throughout the study, the participants studied the lines of code of the test case or the methods under test to understand their behavior or determine the values of test objects. This connects to existing evidence that understanding the setup and input of a test case is central for developer comprehension [92].
- **Method calls:** The participants used the branches of the graph to inspect the behavior of statements that call methods by drilling deeper into the behavior of these methods.
- **Additional coverage:** Through the bright green highlighting and the initial layout showing all lines with additional coverage, some participants focused their analysis on these lines. They analyzed the proposed test case to infer how its statements and the methods they call lead to the execution of the highlighted lines.
- **Inner-method control flow:** Some participants tried to retrace the control flow inside of the methods under test. They made use of the coverage highlights to determine which lines were executed and inferred the outcome of conditional statements.
- **Actual values:** Our participants indicated that access to the actual values of variables and method parameters would help them retrace the behavior of the test execution

even better, similar to debugging a test failure [28]. During our study, the participants manually went through the code of the test case and the methods under test to infer the actual values of variables.

- **Natural language explanation:** A participant wished that the tool should give a textual, more abstract description of the new input provided to the method under test, compared to the input other tests provide to the same method, e.g., "Other tests cover this method with an empty list, this test additionally executes it with a non-empty list".

We saw that different developers take different approaches to understand a developer test. A suitable visualization should provide information that supports many of these approaches.

Recommendation 1A

A developer test visualization should **provide access to the wide variety of information** sought by developers **to *understand* the inspected test case**. Under this fall the test name and the source code, the execution flow between and inside of methods, the values of variables and parameters, as well as the coverage and the high-level behavior of the test case.

3.4.2 Should I Test This? Provide Scope of Where Code Is From

Surprisingly, many of our participants used the TestImpactGraph in a very similar way to common code coverage tools, a popular means of developers to inspire new test cases [28]. They inspected which lines were covered by the test, but also focused on the lines not covered. Several of them pointed out grey lines, which they assumed to be not covered and stated that they would write additional tests for them.

In this context, it is important that the visualization does not mislead the developers, as the grey lines in the TestImpactGraph are simply not covered by the proposed test case.

Recommendation 1B

A developer test visualization that presents coverage information to developers should **be careful when displaying code as not covered**, as developers might use the visualization to identify code to cover in additional test cases.

In addition, some participants asked if certain methods were part of a library or the code of the project itself. They considered it relevant to cover methods of the project itself, but not code that is part of a library.

Similar to this, one participant wished that the test method, i.e., the JUnit method which defines the developer test, as well as any helper methods are visually distinguished from the system code. Their reasoning was that while test helper methods are executed by the test case, they do not need to be covered and would not contribute to describing the additionally tested scenario.

Recommendation 1C

A developer test visualization should **distinguish the *origin* of presented code pieces**: the system under test, dependencies, or the test code. This helps developers judge if a certain piece of code should be tested and if the coverage of this code is relevant for the strength of the test suite.

3.4.3 Who Tests This Already? Support Exploring Other Tests

When presented with dark green highlighted lines in the methods under test, i.e., lines that are already covered by other tests in the test suite, several participants wondered *which other test case* was executing these lines already. They wished for functionality to inspect for each line the set of test cases that covers this line, a kind of line-based code-to-test traceability [93].

There were several reasons for the different participants to seek this kind of *reverse coverage* information [94]: looking at and understanding the other test cases which execute the same lines of code could help the users understand the developer test they currently inspect. When the dark green lines appeared in combination with bright green ones, i.e., indicating that the bright green code is only covered by the inspected test case, the developers wondered about the difference between the test case they currently inspect and the ones already covering the dark green lines. They were interested in why the current test case executed the additional instructions, why the other test cases did not. A different consideration was to minimize either the current or the other test cases, to not unnecessarily cover code parts twice. Similarly, Panichella et al. [35] found that developers would want to be notified if a part of the behavior of a generated test case is already checked in a previous method.

Recommendation 1D

A developer test visualization which indicates that code parts are already covered by another test, should **provide information on which *other* tests already cover a line under test**. This can help the developer to take refactoring decisions to improve the focus of their test cases.

3.5 RQ2: Which Features of the TestImpactGraph Help Developers Access This Information?

To answer our second research question, we collected observations and feedback from the participants to gauge which existing and potential features help them access the information they seek through the TestImpactGraph.

3.5.1 Code Nodes: As Close to the IDE as Possible

We received several points of feedback concerning the method nodes in the TestImpactGraph. The participants inspected the statements in several method nodes closely, e.g., to reconstruct the behavior of a method or determine the value of an object under test. When presenting source code, the users expected syntax highlighting as it would help them *"spot*

variables being reused throughout the method". The full signatures of the methods under test helped them to reconstruct the behavior of a method and the values returned. As our participants were unfamiliar with the example project from which they inspected test cases, they wished for access to the documentation of several methods they encountered in the TestImpactGraph.

Recommendation 2A

Developers expect a **direct presentation of source code** to be **as similar as possible to their familiar IDE environment**. In our study, this included syntax highlighting, the whole signature and source code of the method and access to its documentation.

3.5.2 Default View: Provide Confidence to See Everything Relevant

Showing all code with additionally covered lines right away was a big success with the developers that participated in our study. They were glad not to have to search through the graph for more newly covered lines and that they could focus on what is presented at the beginning. After exploring the graph and opening many branches, a participant wished for an easy possibility to return to the default view, so they could re-focus on the relevant code paths. Others proposed to highlight the methods leading to additional coverage, so they could be distinguished even while other nodes and branches are opened.

Recommendation 2B

A developer test visualization should help the developer **focus on the central and unique parts of the execution of a test case**. This prevents overwhelming the user with less relevant behavior details, e.g., details in the methods setting up test objects.

3.5.3 Where Was I? Providing and Keeping Context

Our participants were thankful for the flat visualization of all methods relevant to the execution of this test case. This let them focus on the methods under test, as well as their connection, better than in an IDE, where they have to switch back and forth between source files to inspect the methods under test.

During our evaluation, we saw that it was important to let the developers maintain their mental context of the nodes visualized on the screen. This includes maintaining the previous layout when new branches are opened and new methods are inspected, even if it required manual zooming and panning from the user to see the new nodes.

Recommendation 2C

The layout of the method nodes in a developer test visualization should **stay consistent while the developer interacts with the visualization**. This lets the user build up and maintain a mental context between the developer test and the code under test.

3.5.4 Where Does This Connect? Clarifying Edges

We also learned that the default layout of the graph contributes heavily to the interaction of the developer with the graph. In the case of our TestImpactGraph prototype, some call edges were overlapping with each other or partially covered by unrelated method nodes, meaning the participants had to move the nodes around to identify which code line was calling which method node. Similarly, there were cases in which the method nodes overlapped, requiring interaction from the user to make all nodes they found relevant visible. The participants wished for a clearer layout that does not require their interaction, giving them more time to focus on the presented test case.

We discuss this and the previous aspects because they confirm existing results from the field of information visualization. In a study about visualizations for software exploration, Storey et al. [90] name the reduction of user effort in adjusting interfaces as an important design element. Guidelines for UML class diagrams [95, 96] recommend to avoid edge crossings or overlapping nodes, to support users in recognizing the presented objects.

We observed another visualization challenge more specific to the TestImpactGraph. As the method calls are connected to a particular line, when multiple methods are called it was difficult to distinguish for a participant which method nodes correspond to which method call. One participant resorted to comparing the names of the methods, noticing that this falls short if two have the same name or one method is called, possibly with different parameter values. One way to address this could be the use of color on the method calls and edges, similar to the "smart step into" feature of IntelliJ[11].

Recommendation 2D

The layout of developer test visualization should **show clearly which method call and method node the ends of an edge connect to** and should not require the interaction of the user to clarify the presented information.

3.6 RQ3: What Observations Related to Test Coverage Arise When Inspecting a Developer Test Through the TestImpactGraph?

Inspecting test cases through the TestImpactGraph gives the developer the chance to take a novel look at the behavior and especially the additional coverage of a developer test. We want to report on a few interesting observations our participants made while using the TestImpactGraph on the three examples we provided.

3.6.1 Should This Not Already Be Unit-Tested? Deep and Accidental Coverage

As shown in Figure 3.2, our third example shows a rather large TestImpactGraph, because some of the additionally covered lines are multiple method calls away from the developer test. Two of our participants noted this and wondered whether these methods should not

[11] https://www.jetbrains.com/help/idea/stepping-through-the-program.html#smart-step-into Accessed: June 1st, 2021

be covered more directly by a unit test, instead of by the more integration-style test they were inspecting. As there was also a whole new method covered directly through a call from the developer test, one participant even wondered if these further methods were covered "accidentally" by a test meant to test the directly called method. The participant said to not trust this accidental coverage and built their answer to what the test is newly testing solely on the directly called and newly covered method.

A participant also pointed to similar issues if a method would lead to additional coverage in several different, unconnected areas of the code under test. This would give them the impression that it is not clear what the test case is really intending to test.

Recommendation 3A

According to our participants, **additional coverage that is several method calls away could point to a lack of unit testing or be considered accidental**. Accidental coverage was deemed less relevant to determine the impact of a test case. We recommend to further investigate whether such accidental coverage should—and how it can—be removed from amplified tests or developer test visualizations.

3.6.2 What Is Executed Here? Instruction Coverage Visualized Per Line

As described before, several of our participants approached understanding the methods under test by going through their statements one by one. Where possible, they made use of the line highlighting that indicates which lines were executed. However, the common practice of visualizing instruction coverage as a highlight of a whole line led to confusion in some cases. In the first example test case (see Figure 3.4), there are three one-line `if` statements with `return` statements in their bodies. All three lines are highlighted in bright green, i.e., indicating that there were additional instructions covered on these lines. The participants were confused how all of these `return` statements could be executed in one method call.

While there are newly covered instructions on these lines of code—namely the conditions of the `if` statements—the highlight's indication that the whole line is executed was confusing. A participant reflected, that it would be better to indicate that while additional instructions are covered, not all instructions on these lines are covered by the inspected test case. They added that this would also depend on the code style of the code under test, e.g., whether the `if` and `else` blocks of a conditional statement are on separate lines from the conditions.

We made a similar observation in the second example test case (Figure 3.1), where seemingly a whole new method is covered: all lines in the method are bright green. The method only consists of a `for` loop with statements in it. However, our participants wondered why the statement calling this method was dark green, indicating that it was already executed by another test case. Upon closer inspection, we determined that the previous callers executed the method in question with an empty list, therefore the first instruction of the `for` loop was executed. In the new, inspected test case, the method in question was called with a filled list, leading to the execution of the statements in the `for` loop, as well as the increment statement in the `for` loop's header. A participant noted

that it would help to indicate that some of the instructions on the header line are already executed by other test cases.

Recommendation 3B

When instruction coverage is visualized at the line level, it should be indicated whether **instructions on a line were previously covered**, and whether **there are still uncovered instructions on a line**.

3.7 Discussion

Our exploratory think-aloud study resulted in a range of 10 actionable recommendations regarding the information developers seek from the TestImpactGraph, features to help them access this information and observations based on the new view angle on additional coverage of developer tests. Several of them confirm existing knowledge from information visualization (**2A**, **2C**, **2D**), extend existing intuitions for the area of developer test inspection (**1A**, **1C**, **2B**), and others point to novel challenges special to developer test inspection (**1B**, **1D**, **3A**, **3B**). In this section we discuss why our results can also be applied to the area of test code review, due to the large similarities between inspecting a test case for amplification and for test review. Furthermore, we illustrate how the TestImpactGraph can provide insights that lead to a more fine-grained coverage understanding related to test directness and redundancy.

3.7.1 Test Review

The review of test code during traditional code review has many parallels to the inspection of proposed test cases during test amplification. In both cases, developers judge whether a new developer test is adequate and should be included into the test suite. Our results also show these parallels to previous work on test review by Spadini et al. [59]. Similar to our observations that developers use the TestImpactGraph as a coverage tool to spot uncovered lines (Section 3.4.2), a central concern in code review is to understand whether the test covers all paths [59]. None of our participants asked whether they could inspect the presented code in their IDE, a typical behavior of developers to gain a complete picture of the code during test review [59]. Therefore, we hypothesize that if test reviews are performed inside the TestImpactGraph tool, Spadini et al.'s recommendation to provide better navigation between the developer test and the code under test would be addressed.

3.7.2 Differential Code Coverage

A tool dedicated to surface coverage changes during code review is Codecov[12]. It provides high level, aggregated coverage information about the whole project, but also *differential coverage* introduced by a commit or a pull request. Their differential coverage indicates how much of the code diff is covered an how the change impacts the overall project coverage. In their source code view[13], developers can inspect the code under test enriched with highlights that show how the coverage of single lines change through the inspected

[12] https://about.codecov.io/
[13] https://docs.codecov.io/docs/viewing-source-code

commit or pull request. While Codecov provides a coverage diff for any kind of code change, i.e., to the test code and the code under test, the TestImpactGraph focuses on the addition of one test case, while the code under test stays constant. Furthermore, the TestImpactGraph is showing the method call connections between the developer test and the methods under test, letting the developer retrace the execution of the test case. This addresses an important point in test coverage evolution: helping developers understand better why a change in the code leads to a change in coverage [97].

When inspecting a test case that is proposed to amplify a test suite, or an added test case in traditional code review, a tool like Codecov would give quick feedback on how the test case impacts the coverage of the test suite, while the TestImpactGraph would give more detailed insights to the developer on where and why the test case impacts the code coverage.

3.7.3 Refined Coverage Insights

From our observations about already covered code (**Recommendation 1D**), as well as deep and accidental coverage (**Recommendation 3A**), we see a chance for the TestImpactGraph to give developer's a deeper insight into how their tests cover the code under test. The reflections of the participants show that coverage could be interpreted as more than just covered or not covered. Instead, the participants also considered how directly test cases are covering a specific method. Test directness is important to help developers pinpoint the fault in the code when a test is failing [38]. It also impacts how well a test case can serve as documentation of the code under test [43].

Whether these insights lead developers to adapt their test suite to be more direct, will depend on the needs of the software project and their testing culture, e.g., the relative value of unit and integration tests for the project. Independent from the testing culture of a project, the TestImpactGraph can give the developers deeper insight into the coverage that single test cases contribute, enabling them to take informed decisions about the design of their test suites.

3.7.4 Relevance of Deep Coverage for Test Descriptions

Current techniques to automatically generate names for unit test cases use, beneath other information, the names of the methods executed by a test case as basis for the generated names [32]. The automatic test generation community is moving towards generating integration tests that check the interaction of multiple classes [98–100], and the amplified test cases from our study were also integration test that executed several methods. Based on the feedback we got regarding deep and accidental coverage (Section 3.6.1), the question becomes whether methods that are executed, but further away from the developer test in the call chain, are relevant to generate test names that are meaningful for developers. Similarly, we should investigate, whether or how deep coverage can be used for natural language test descriptions, such as those generated by Panichella et al. [35] or Roy et al. [34].

3.7.5 Threats to Validity

In the following, we discuss several threats to the validity of our results.

With regards to construct validity, the amplified test cases we chose for our study might be favorable for the TestImpactGraph. Long, complex test cases that additionally cover a large part of the source code would be difficult to inspect with the current design. Similarly, for simple test cases it might feel unnecessary to use more than the tools available in an IDE. To mitigate this threat, we picked a variety of test cases that represents the types of coverage we see when amplifying test suites with TestCube. All the tools we used[14], as well as our experiment data[15] are openly available and we encourage others to explore them to retrace our observations.

We assume a broad target audience for the TestImpactGraph, novices as well as experienced software developers. However, our participants might not match this audience, as we used convenience sampling to select participants. The reflective insights about test directness and accidental coverage might stem from the fact that our participants are mainly PhD students and therefore more aware about software quality than other software developers. Future work is needed to investigate how professional experience impacts developer's interaction with the TestImpactGraph.

As we used convenience sampling, the researcher and the participants knew each other personally, however the participants had not been involved in the researcher's work before the study. The participants were motivated to participate to support the author's research work and could have been biased to give more positive feedback during the study. We mitigated this threat by emphasizing the value of critical and opinion-rich comments, and focusing our results on the concrete recommendations rather than ratings of the existing design.

A threat to the external validity of our study might be the underlying test amplification approach we used to generate the test cases for our study. The TestCube tool selects which test cases to propose based on whether they cover additional instructions in the code under test. One could choose other selection criteria, e.g., whether a new edge case is checked, which would call for another kind of visualization, e.g., because no additional instructions are covered. Another threat is the selection of the example project, or the test cases we inspected. We aimed for a middle-sized project and a variety of test cases with respect to their coverage characteristics. This lead to a variety of visualization in the TestImpactGraph, and let us widely explore visualizations from an average Java project. We do not claim our findings to be applicable to every software project, and their relative importance will vary depending on the developer interacting with our tool.

3.8 Related Work: Test Visualizations

In this section we present related scientific works in the area of visualizing software test cases or associated metrics.

Visualizations are used to judge the quality of a whole test suite, visualizing test metrics, including code coverage, in conjunction with the system under test. Borg et al. [101] visualize historical test outcomes in a code city of the system under tests. Their goal is to help identify error-prone components and potential coverage holes, i.e., components that are not covered enough by the test suite. Balogh et at. [102] extend a different code

[14] https://github.com/TestShiftProject/test-cube/tree/test-impact-graph, https://github.com/TestShiftProject/test-impact-graph

[15] https://doi.org/10.5281/zenodo.6644723

city framework to visualize test-related metrics together with the components of the system under test. Perscheid et al. [103] use a variety of tree maps, presenting different quality aspects of a test suite. They aim to help developers pinpoint untested code and improve time or memory intensive test cases. Opmanis et al. [104] present a dashboard that visualizes the test results of large systems, helping managers and quality engineers to identify the origin of deteriorations or improvements. The TestQ tool by Breugelmans and Van Rompaey [105] presents a tree-like overview over the structure of a whole test suite. For a closer inspection, singular test cases are presented as a hierarchical structure of the test components like fixture and test helpers, annotated with instances of test smells. A separate window provides an overview over all smells appearing in the test suite and lets the developer spot very smelly tests. While these visualizations aim to give an overview of the whole test suite, the TestImpactGraph is geared towards visualizing the execution of a single test case and visualizing it coverage, focusing on the impact it makes on the coverage of the whole test suite.

Other approaches visualize the current coverage of a test suite to aid developers in achieving better coverage. Vanessa Peña Araya [106] proposes *Test Blueprints*, a hierarchical visualization of the structure of the code under test in conjunction with how often its methods are executed by the test suite. Lawrance et al. [88] highlight lines in the code under test to indicate the existing quality of a test suite and investigate whether this leads developers to create more effective tests. With a similar goal, Rahmani et al. [107] create a control flow graph showing the current branch coverage. Among our participants, we observed a similar, intuitive strive to write additional test cases for lines that were not marked as covered.

Previous work investigated visualizing the software components, methods or lines that are executed by a test case or an interaction with the system to aid understanding of the behavior of the system under test. Cornelissen et al. [108] visualize tests as abstracted scenario diagrams, Arthur-Jozsef Molnar [109] visualizes the components executed by a sequence of GUI interactions, and Gestwicki and Jayaraman [110] present the structure of live object and values as well as method invocations during the execution of a java program.

The research area of test-to-code traceability is similar to our effort to connect a developer test to the code it tests. The IDE extension EzUnit by Bouillon et al. [111] creates links to jump from a test case to the methods under test. Aljawabrah et al. [112, 113] visualize the connection between unit tests and the code under test with the TCTracVis tool. Their tools shows traceability links in a hierarchical tree graph in both directions: from test to code and from code to test. While TCTracVis is built to support different methods to retrieve traceability links and visualizes the connection of high-level code structures like classes and methods, the TestImpactGraph relies purely on coverage information and visualizes the connection of test and code under test on a line granularity. Furthermore, the TestImpactGraph presents the source code directly to the developers, keeping the visualization as close as possible to the source code [81].

Vidaure et al. [114] use visualizations to shed light on the internal processes of automated test generation. Their tool TestEvoViz lets developers examine the process behind a genetic algorithm to see the impact their configuration has on the test generation. While the TestImpactGraph also works with automatically generated test cases, its design is in

principle independent from the specific test generation approach. The TestImpactGraph steps in after the test cases are generated, with the aim to help the developer inspect one specific new test case at a time.

3.9 Conclusion and Future Work

In this chapter, we present the TestImpactGraph, a visualization of developer tests to aid understanding of their behavior and impact. Through an exploratory think-aloud study we identify a range of ten recommendations for future visualizations of developer tests.

We discuss how the TestImpactGraph can give developers a better context of the code under test during code review, how it complements differential coverage and how developers can use it to gain a more fine grained understanding of the coverage their test cases provide. In future work we aim to apply the recommendations we presented to the TestImpactGraph itself and study in-depth how much it helps developers explore automatically generated test cases, as well as test cases written by their colleagues. In addition, we intend to compare how understanding test cases with the specialized TestImpactGraph compares to using standard debuggers, IDE features like *jump-to-definition*, or general-purpose visualizations of the tests' call graphs. We want to investigate other means to convey the behavior and impact of a test case, e.g., through test names or textual descriptions, leveraging our observations about deep and accidental coverage. Further, we will explore ways to help developers understand test cases that improve the test suite in other ways than instruction coverage, such as covering edge cases, reproducing known bugs or killing artificial mutants of the system under test.

In short, this chapter contributes:

- The design and a prototypical implementation of the TestImpactGraph, a visualization to help developers understand the behavior and impact of a test case generated to amplify an existing test suite.
- Ten actionable recommendations on how tools with a similar aim should visualize developer tests.

Our vision is that tools like the TestImpactGraph will enable developers to dive deep into new test cases and help them understand how and why these test cases amplify the power of their test suite.

4

Shaken, Not Stirred. How Developers Like Their Amplified Tests

Test amplification makes systematic changes to existing, manually written tests to provide tests complementary to an automated test suite. We consider developer-centric test amplification, where the developer explores, judges and edits the amplified tests before adding them to their maintained test suite. However, it is as yet unclear which kind of selection and editing steps developers take before including an amplified test into the test suite. In this chapter we conduct an open source contribution study, amplifying tests of open source Java projects from GitHub. We report which deficiencies we observe in the amplified tests while manually filtering and editing them to open 39 pull requests with amplified tests. We present a detailed analysis of the maintainer's feedback regarding proposed changes, requested information, and expressed judgment. Our observations provide a basis for practitioners to take an informed decision on whether to adopt developer-centric test amplification. As several of the edits we observe are based on the developer's understanding of the amplified test, we conjecture that developer-centric test amplification should invest in supporting the developer to understand the amplified tests.

This chapter is based on *C. Brandt, A. Khatami, M. Wessel and A. Zaidman. Shaken, Not Stirred. How Developers Like Their Amplified Tests, IEEE Transactions of Software Engineering, 2024* [54].

Automated testing has become central to ensure a high quality during software development [28, 38, 58]. Nevertheless, writing tests is seen as a tedious and time-consuming task [6–8]. This is where automatic test generation comes in by supporting developers and relieving them of the burden of writing tests [10, 46, 115–117].

State-of-the-art test generation tools are powerful in protecting against regressions [20], finding crashes [118], and reproducing crashes [22, 23]. However, they are rather difficult to adopt in day-to-day software engineering, in part due to the difficulty to understand the generated test scenarios [45], [32]. For developers it is crucial to understand a test when it fails and they have to localize the underlying fault [35], [41].

This is where *test amplification* shows promise: instead of generating completely new tests, e.g., with genetic algorithms (e.g., EvoSuite [10]), test amplification makes systematic changes to existing, manually written tests with the intent to provide tests that are complementary to the existing test suite [42]. In contrast to generated tests that are stored separately from manually written tests, e.g., when tests are regenerated after software evolution [21, 37], our focus is on *developer-centric amplified tests*. Developer-centric test amplification is a concept we coined in our previous work (Chapter 2). It proposes that developers adopt the amplified tests into their main test suite, potentially after manually adjusting the amplified tests. Developer-centric test amplification means (1) developers benefit from only having to validate amplified tests, instead of writing these tests manually, and (2) understanding the tests should be easier because they originate from manually written tests. To illustrate this more vividly we introduce an example use case of developer-centric test amplification:

> Adriana is a software developer in a project that is struggling with automated testing, as pressure for new features makes it hard to find time to write tests. She has some time left this sprint and decides to invest it into testing. To be quicker, she uses a developer-centric test amplification tool which generates compiling and passing tests that cover code that is not covered by the test suite. Adriana browses through the proposed tests, inspecting their behavior and new coverage contribution to judge which ones to include in the test suite. Whenever she decides to keep a test, she takes a look at its code and does some adjustment to make them easier to understand for her colleagues and fit better to their project's style and quality. After adding several new tests into the test suite of her project, she commits them all and prepares a merge request that describes the improvements to the test suite.

While several studies have investigated the shortcomings of generated and amplified tests from the developer's perspective [45, 47], (Chapter 2), little is known about which kind of adjustments developers would make to an amplified test before including it in the test suite. Therefore, the goal of this chapter is to better understand the effort that developers need to go through when (1) deciding whether to add an amplified test to the test suite, and (2) adjusting the amplified test before it can be added. To this end, we conduct a qualitative open-source contribution study [13, 119]: We amplify tests for 52 open-source projects and open 39 pull requests to contribute the amplified tests back to the projects. For the test amplification, we employ DSpot, which is the original, arche implementation of test amplification for Java created by Danglot et al. [13, 42]. Our qualitative investigation in this chapter is steered by the following research questions:

RQ1: What deficiencies do we observe in DSpot amplified tests when preparing them for a pull request?

RQ1.1: On which criteria do we select a candidate test to include in the test suite?

RQ1.2: Which manual edits do we perform to improve the tests before submission?

RQ2: What feedback do we receive from the maintainers on the DSpot amplified tests?

RQ2.1: Which changes are proposed during the pull request discussion?

RQ2.2: What kind of information is requested by the maintainers during the pull request discussion?

RQ2.3: How do the maintainers justify their judgment over the amplified tests during the pull request discussion?

Based on an existing dataset of buildable Java repositories [120], we try to amplify tests for 312 open source projects. We employ the developer-centric test amplification of DSpot [18], (Chapter 2), together with a new automatic post-processing module that filters and simplifies the amplified tests. For each of the 52 projects where the test amplification succeeds, we manually select a candidate test to submit in a pull request. The criteria that emerge during this selection process answer **RQ1.1**. We manually edit the candidate tests to improve their quality before opening a pull request. Based on our experiences in this phase, we build a checklist of edits to expect, the answer to **RQ1.2**. To validate whether these edits would also be proposed by open source maintainers, we omit the manual editing for half of the projects.

We open pull requests for 39 projects with one amplified test each. To clarify our contribution to the project maintainers, we provide an automatically generated textual description of the amplified test. During the discussion, we incorporate any proposed changes and answer arising questions. 19 pull requests were accepted and 13 closed. We analyze the discussions on the completed pull requests to elicit the changes that the maintainers propose (**RQ2.1**), the information they request to understand the amplified tests (**RQ2.2**), and how they justified their judgment over the amplified tests (**RQ2.3**). As we manually selected which amplified tests to submit and manually edited half of them to improve their quality before submitting, the results for the second set of research questions more closely represent what amplified test are capable of with human intervention, or with automation advancing might be capable of in the future.

4.1 Developer-Centric Test Amplification

The technique of *test amplification* generates new tests by modifying test that were written by developers [42]. Our work is based on the developer-centric test amplification of DSpot [13], (Chapter 2), which we introduce in this section.

To explore new behavior, DSpot mutates the setup and action phase of an existing

```java
// Together with all generated tests, reaches a mutation score of 0.8518
@Test public void test09() {
 InputStream.nullInputStream();
 PipedReader pipedReader0 = new PipedReader();
 Reader reader0 = Reader.nullReader();
 ByteArrayOutputStream byteArrayOutputStream0 = new ByteArrayOutputStream(17);
 ObjectOutputStream objectOutputStream0 = new ObjectOutputStream(byteArrayOutputStream0);
 BufferedOutputStream bufferedOutputStream0 = new BufferedOutputStream(objectOutputStream0);
 MockPrintStream mockPrintStream0 = new MockPrintStream(bufferedOutputStream0, true);
 CopyUtils.copy(reader0, (OutputStream) mockPrintStream0);
 Reader reader1 = Reader.nullReader();
 assertNotSame(reader1, reader0);   }
```

Figure 4.1: Test generated by EvoSuite for apache/commons-io.

4

```java
// Covers new instructions in ByteArrayOutputStream.reset and AbstractByteArrayOutputStream.resetImpl
@Test public void testToByteArrayImplAndResetImpl() {
 InputStream in = new ByteArrayInputStream(inData);
 in = new ThrowOnCloseInputStream(in);
 final ByteArrayOutputStream baout = new ByteArrayOutputStream();
 final OutputStream out = new ThrowOnFlushAndCloseOutputStream(baout, false, true);
 final Writer writer = new OutputStreamWriter(out, StandardCharsets.US_ASCII);
 CopyUtils.copy(in, writer);
 writer.flush();
 baout.reset();
 Assertions.assertEquals("", baout.toString());   }
```

Figure 4.2: Test generated by developer-centric DSpot.

test, called the *original test*, by changing the values of literals and removing or adding method calls to the objects under test. The old assertions are removed and replaced by new assertions. For the oracle, DSpot uses the current behavior of the system: it executes the test and observes returned values, which it uses as the expected value of the new assertion. This leads to all generated tests passing. The developer-centric variant of DSpot aims at generating concise and simple tests, so it adds one setup mutation and one assertion per test it generates. Lastly, only tests that execute instructions not yet covered by the test suite are kept and shown to the developers[1].

As the next step in developer-centric test amplification, a developer browses and inspects the new, amplified tests. They judge whether a test is valuable to include into their test suite, e.g., because of the additional coverage it provides. The developer can also edit the tests where they see fit, like adding meaningful names or explanatory comments. The goal is that they include the selected and edited tests into their test suite and keep maintaining them in the future.

Developer-centric test amplification is one instance of a variety of approaches to automatically generate xUnit tests. In comparison to, e.g., the widely studied search-based test generation of EvoSuite [10], it differs in these central points:

1. EvoSuite generally works without input of manually written tests, while DSpot mutates existing, manually written tests [13]. This introduces the assumption of more readable

[1]DSpot can select tests based on mutation score, the developer-centric variant selects on added instruction coverage for its easier explainability and better performance.

tests from the outset.

2. EvoSuite generally aims to generate a whole test suite at once [17], while DSpot's approach is closer to test augmentation: Complementing an already existing test suite with matching additional tests [121, 122].

3. The developer-centric variant of DSpot sees the developer judging and editing a test as a central component before adding the test to a maintained test suite. That is why it should always be combined with additional information and approaches to facilitate the communication between the test generation and the developer (Chapter 2).

Figure 4.1 and Figure 4.2 illustrate the difference of tests generated by EvoSuite and developer-centric DSpot, such as the variable names and the information given about the impact of the generated test (see line 1).

Recently, Roslan et al. [123] extended EvoSuite to support test amplification in combination with EvoSuite's powerful search-based test optimization. While they reported anecdotal evidence of less readability than DSpot-generated tests, all previous developer-involving studies with EvoSuite do not consider the test amplification approach. In Section 4.6 we connect and contrast our findings with those of the previous user studies of EvoSuite.

Another approach that can be related to test amplification and search-based test generation is *fuzzing*, where random, but valid inputs are generated and iteratively mutated to test the robustness of a software system [5]. While the techniques overlap in their use of mutation and aim to improve the quality of the software under test, there are significant differences that make it difficult to apply the findings of developer-centered fuzzing studies [124, 125] to our work. Fuzzing focusses on highly structured test inputs and requires the use of fuzzing harnesses to call the system under test [126]. In comparison, test amplification and search-based test generation produce ready to use test structures leveraging xUnit frameworks [127], which developer-written tests also use. Furthermore, fuzzing primarily targets robustness, aiming to uncover crashes or unintended execeptions in the software under test [5]. Because of this, fuzzing is often used to address security and reliability concerns, where any fuzzer output that leads to an undesirable crash is relevant to be addressed [128]. In comparison, developer tests like the ones produced by test amplification typically have a functional oracle or assertion that checks that the code under test behaves as expected. Therefore, the tests generated by amplification and search-based approaches improve the quality of the functional test suite, which in turn improves the confidence in the correct behavior of the code under test. Beyond that, the developer test suite can also serve as documentation [38–40] and a starting point for developers to localize the root cause of a test failure [35, 41], two use cases where the understandability of the tests is crucial.

4.2 Automatic Post-Processing for developer-centric test amplification

We previously conducted an exploratory study to evaluate a test amplification plugin for the IntelliJ IDE (Chapter 2). The developers we interviewed mentioned several aspects they would change before accepting the amplified tests into their test suite. For example,

removing unnecessary statements or changing cryptic identifiers to meaningful ones. The participants also pointed to methods that they found not relevant to test, e.g., simple getters. To automate these already known points, we design an automatic post-processing tool for developer-centric amplified tests: the *prettifier*. The prettifier is based on an existing module in DSpot and is run after the amplification described in Section 4.1. The aim of the prettifier is to make the resulting tests: (1) more concise, (2) easier to read, and (3) more relevant to developers.

The participants of the previous study spent a lot of their time *understanding the behavior* of an amplified test (Chapter 2). This understanding was the basis for their judgment on whether to accept a test into their test suite. Previous studies have shown that a natural language description helps developers to understand generated tests [34, 35]. To reduce the effort required by developers to understand an amplified test, we generate natural language descriptions of the behavior and impact of the test compared to the rest of the test suite.

In this section we will present our design for the prettifier and the description generation for amplified tests.

4.2.1 Prettifier module

To automate several of the post-processing steps indicated by our previous study (Chapter 2), we extend Danglot et al.'s prettifier module for DSpot [18]. Our approach takes three steps: (1) minimizing the tests to make them faster to read, (2) renaming variables and the test methods to make them less cryptic and more expressive, and (3) filtering and prioritizing the tests according to their relevance to the developer.

Minimizer

To remove statements that were part of the original test, but are not relevant for the amplified test, we adopt Oosterbroek et al.'s approach [71]. They minimize amplified tests, while retaining the provided additional coverage. The approach works in increasingly conservative steps: a) remove all statements except the assertion and the ones needed for the code to compile, b) remove all statements that do not directly interact with the assertion, i.e., by setting variables used in there, or c) remove all statements that do not (in)directly interact with the assertion, i.e., by calling a method on the object involved in the assertion. When a step decreases the coverage or causes the test to fail, the next step is tried.

We also activate two existing minimizers of DSpot. One in-lines single use variables created by the DSpot amplification, the other removes redundant casts included by the amplification for safety.

Test and Variable Renamer

To make the tests easier to read and understand, we implement a simple variable renamer that hides DSpot's intermediate variable names (`__DSPOT_path_696`) with less cryptic, simple names (`String2`, pattern: `<Type><N>`). Further, we generate meaningful names for the amplified tests based on the additional coverage they provide using the NATIC approach [33]. NATIC identifies in which unique methods a test covers additional instructions, compared to the other amplified tests and the existing test suite. Similar to Daka et

al.'s approach [32], we rank the methods according to how much additional coverage they contain, concatenate up to two of the method names and generate a unique test name such as "`testGetFileAndHasLength`".

Filter and Prioritize

One issue with automatic test generation can be the large number of tests produced. Specifically with developer-centric test amplification, some generated tests target methods that developers find irrelevant to test, such as simple getters, or `hashCode`. To reduce the number of tests not relevant to developers, we included a developer-centric filter in the prettifier. It removes tests that only contribute coverage in simple getters or setters, i.e., one line methods starting with "get" or "set". The filter also removes tests that only add coverage in Java's `hashCode` method. Because exception handling code is commonly under-tested [129, 130], we explicitly keep any test that checks for an exception. The prettifier puts the test with the most additionally covered instructions first, so that the developers inspect the most impactful amplified test first.

4.2.2 Descriptions for Amplified Tests

In our previous study (Chapter 2), we saw that a major step for the developers was understanding the behavior and intent of an amplified test. The developers studied the code of the test, compared it to the original test and inspected the newly covered code under test. To support the understanding of amplified tests, we design an approach for an automatically generated, natural language description for amplified tests. The description surfaces the behavior and impact of the test compared to the existing test suite. It is meant to be informative for the developer without having to read the code, e.g., as a description in a pull request that proposes an amplified test.

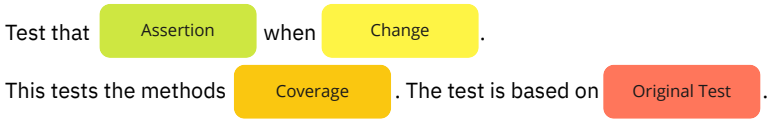


Figure 4.3: The basis template for our description of amplified tests.

Similar to previous test description generators [34, 35], we use a template-based approach. It consists of four components, as presented in Figure 4.3: (1) Describing the assertion, (2) describing the change to the setup of the test, (3) describing the additional coverage that is contributed, and (4) pointing to the original test. We fill these components based on information collected during the amplification process. Figure 4.4 shows an example test and its corresponding description. In this case, the **assertion** is an expected exception, the **change** made by the amplification was to set the value of a literal method call parameter to an empty string. The description indicates that additional **coverage** is situated in the method `BuilderFactory.build`, and that the **original test** was `buildDouble`. The full templates and our implementation are open-source and shared as part of our replication package [131].

```
Test that a java.lang.NumberFormatException is thrown
when the parameter data is set to "".
This tests the method BuilderFactory.build.
This test is based on the test buildDouble.
```

```
/* Coverage improved at
redis.clients.jedis.BuilderFactory.build L. 8 +2 instr. */
@Test public void testBuild() throws Exception {
 try {
  Double build = DOUBLE.build("".getBytes());
  fail("testBuild should throw NumberFormatException"); }
 catch (NumberFormatException expected) {
  assertEquals("empty String", expected.getMessage()); }}
```

Figure 4.4: An example amplified test and its generated description. The original test and the name of the changed parameter are not visible.

4

4.3 Open Source Contribution Study

The goal of this chapter is to gain a clearer understanding of the changes that developers would make to amplified tests before including them into their test suite. To this end, we conduct a qualitative open source contribution study [13, 119], utilizing DSpot's developer-centric test amplification, our improved prettifier, and the automatically generated descriptions for amplified tests. The central step of the contribution study is to open pull requests with amplified tests to open source projects. However, it was crucial to us to not antagonize the project maintainers against us or the research community [132], [119]. Thus, we first carefully selected amplified tests that we believe are a valuable contribution to the project, and only opened a pull request if we found any. We document the criteria that arose during this selection process, including how often we applied each of them (**RQ1.1**). We also received feedback on the value of the submitted tests during the pull request reviews, which we analyze to answer **RQ2.3**.

As the maintainers of a software project are responsible to update the tests when the software evolves, their feedback is invaluable to understand which changes are necessary before including an amplified test in a maintained test suite. This is why analyzing the changes proposed during the pull requests is a central part of our study (**RQ2.1**). To keep the burden on the open source developers as minimal as possible, we manually edited and improved the amplified tests for half of the projects before submitting the pull requests. The other half we submitted without editing, to validate whether the edits we choose would also be proposed by a maintainer. To lead our editing, we created and continuously updated a checklist of potential edits, which we use to answer **RQ1.2**.

Another ambition of our study is to evaluate whether the automatically generated textual descriptions are helpful for understanding the behavior and value of amplified tests. Therefore, for a third of the projects we submitted the pull request with the generated description. For another third, we submitted the description and a question on whether the explanation was helpful, and for a third of the projects we submitted the pull request without any explanation of the amplified test. When analyzing the pull request discussions, we study what kind of information the maintainers requested, and the connection to whether an explanation was provided initially (**RQ2.2**).

Our qualitative study consists of five steps: First, (1) we select candidate projects for our study. Next, (2) we use the developer-centric test amplification of DSpot and our prettifier

to generate the amplified tests and their descriptions. Then, (3) we manually select and improve the amplified tests, documenting our emerging criteria. After this, (4) we open pull requests with the amplified tests. Finally, (5) we analyze the feedback from the project maintainers during the pull request discussions. In the following, we will detail the separate steps of our study.

4.3.1 Repository Selection

Our first step is to find GitHub projects that are suitable for applying DSpot's test amplification. As our approach requires building Java projects, and selecting coverage-improving tests with the JaCoCo[2] tool, we use Khatami and Zaidman's dataset [120, 133]. They tried to automatically build and calculate the code coverage of 1454 popular Java GitHub projects. We consider the 312 projects for which JaCoCo could successfully measure code coverage, and select one module per project[3].

4.3.2 Running the Test Amplification

We run DSpot on all selected project modules with a budget of 30 min on a desktop PC. For the exact configuration of DSpot and the hardware specification, we refer to our replication package [131]. We collect all test classes generated by DSpot. We also kept partial results, so if the amplification of all test classes would take longer than 30 min we consider all test classes that were completed within 30 min. Next, we apply the prettifier to simplify and filter the amplified tests and generate matching descriptions.

4.3.3 Manual Selection and Editing

We analyzed all amplified tests and created two checklists:

- How we select the best test to submit to the project.
- Which aspects we manually edit to improve the tests before proposing them to a project.

The first two authors reviewed all, the other authors a subset of the tests. Then we met up to come to a negotiated agreement [134] on the points for both checklists. During the selection and editing process of the study, performed by the first author, new points emerged. We validated them through discussions with other authors to mitigate bias and increase the reliability of the checklists [134].

For each project we selected one test to contribute in a pull request: a test we found the most valuable for the project, or a test where we were curious about the maintainer's reaction. For one half of the projects we manually edited the tests with the help of our checklist and own software engineering experience. To validate if such edits are necessary, and understand which edits are important to developers, we left the tests for the other half of the projects unedited. One goal of this study is to contribute to the open source community while learning from their feedback. It was crucial to us to only ask for the community's reviewing effort if we think a test is valuable for the project. If we did not find a test that seemed valuable, we excluded the project from the rest of the study.

[2] https://www.jacoco.org/jacoco/index.html, visited August 2022.

[3] Alphabetically the first. In trials we saw that the amplification not succeeding in one module of a project often means the same for other modules.

lacinoire commented on Jun 15, 2022 • edited Contributor

Hey 😊
I want to contribute the following test:

Test that a `org.eclipse.lemminx.commons.BadLocationException` is thrown when the parameter `line` of `Position.<init>` is set to `-1`.
This tests the method `TreeLineTracker.getLineInformation`.
This test is based on the test `testEmptyDocumentInc`.

Curious to hear what you think!

(I wrote this test as part of a research study at TU Delft. Find out more)

Figure 4.5: An example pull request description from P14-PDM.

4.3.4 Contributing Back the Tests

We opened pull requests for the resulting tests. The pull request description mentions that we want to add a test and the generated description. As Figure 4.5 shows, we modified each mention of a method in the "Coverage" and "Original Test" parts to be a clickable link to the corresponding code on GitHub. The description contains a note that this pull request was part of a research study. However, we did not reveal that the tests were partially automatically generated. This is because we wanted to avoid negative backlash based on biases against automatic test generation. Before opening the pull request, we studied the contribution guidelines of the project and followed them, e.g., validating that a linter passes, or applying an auto-formatter. After opening the pull requests, we answered all questions by the maintainers and incorporated any changes they requested.

4.3.5 Data Analysis

We performed open and axial coding procedures [135] on the pull request discussions completed as of 19-02-2023. The first author analyzed the discussions by inductively applying open coding, wherein they identified discussion points on code changes, requests for information, judgment statements over the tests, and other possibly relevant characteristics of the pull request. They then performed an initial analysis to group the open codes, employing constant comparison [136] to the pull request discussions to validate our interpretation. To increase the reliability of the results and mitigate bias, the first and second authors refined the code set by merging codes together, updating code names, and identifying a different granularity level for a code. The authors discussed the emergent codes together with the original data and modified the codes until they reached a negotiated agreement [134]. The outcome was a set of higher-level categories as cataloged in our codebook [131]. The resulting higher-level categories are structuring the answers to our research questions in the following section, marked in **bold**. The lower-level codes captured the details that we use to illustrate the presented categories by giving concrete examples from the pull request discussions.

4.4 Results

In this section we discuss the results of our study: the test amplification, the manual preparation for the pull request, and our analysis of the discussions with the maintainers. To clarify in which projects each observation occurred, we use shorthand references in the style `P[n]-(E|P)(D|N)(M|C|O|D)`. The number uniquely identifies each project in our study, while the last three characters give a concise overview on the central dependent variables for the pull requests: (1) was the test **E**dited or **P**lain from the amplification tool, (2) did we provide the generated **D**escription or **N**ot, (3) the outcome of the pull request: **M**erged, **C**losed, n**O** reaction yet, under **D**iscussion. Projects where we did not select any test to contribute are indicated as `P[n]-N--`. Table 4.1 gives an overview of all open source projects in our study, including the number of our pull request. We also report the project's size, total number of commits, number of contributors, number of pull requests, and the year the repository was created, showing that our study includes a diverse set of projects.

4.4.1 Running the Test Amplification

The base dataset [120] identified 312 repositories with in total 1821 Java modules that JaCoCo can automatically calculate coverage for. After selecting one module per repository, we in total tried to generate amplified tests for 312 modules. From the DSpot amplification, we obtained 238 classes with generated tests for 62 projects. For the other projects, DSpot crashed during the execution or could not produce any tests that improve the instruction coverage within the budget of 30 min. To these tests we apply the prettifier, resulting in 190 classes with 1297 generated tests for 52 projects. For the gap of 10 projects, all amplified tests were filtered out according to the criteria we explained in Section 4.2.1. Many projects only have a few tests generated (less than 5 generated test in 25 out of 52 projects), with a few large outliers (P51-PDC: 618 tests, P50-PDM: 123, P10-EDM: 96).

4.4.2 RQ1.1: On which criteria do we select a candidate test to include in the test suite?

For each project that we generated amplified tests for, we explored the new tests to choose a candidate test for the pull request. Initial exploration showed that there was a considerable number of unsuitable tests that could not be submitted for a variety of reasons. To transparently show the effort required to select the amplified tests in our study, we document our process of identifying the candidate test extensively. Through this process arose two checklists: one with *negative selection criteria* and one with *positive selection criteria*. With the negative criteria we identify tests that are not worth continuing with, e.g., because they would take so much effort to improve, that writing a new test from scratch felt easier. As we only submit one test per project, we used the positive criteria to pick which test of multiple possible candidates to choose for this study.

Negative Selection Criteria

We excluded amplified tests for the following reasons:
Coverage False Positive (P22-N–, P29-N–, P32-N–, P13-N–,
P34-ENO, P43-N–): Appearing in six projects, the most-prevalent criterion to reject a test was a *coverage false positive*, i.e., tests where inspection revealed no additional coverage

Table 4.1: The open source projects used in our study, including metrics to show their size, activity and age. Metrics were collected through the SEART GitHub search [137] on 2023-09-14. Each pull request can be accessed by the hyperlink in the third column, or via https://github.com/<project>/pull/<pr number>.

ID	Project	Our Pull Request	Lines of Code	Commits	Contributors	Pull Requests	Creation Year
P1-EDC	apache/commons-io	#358	55k	4323	97	478	2009
P2-EDC	apache/curator	#418	57k	2817	114	478	2014
P3-PDC	apache/guacamole-client	#731	118k	6616	80	909	2016
P4-EDM	apache/httpcomponents-core	#349	82k	3742	65	424	2009
P5-ENM	apache/unomi	#436	78k	2608	44	645	2015
P6-N–	apache/zookeeper	–	182k	2511	191	2056	2009
P7-EDM	authme/authmereloaded	#2562	69k	4131	111	740	2013
P8-PDM	axonframework/axonframework	#2244	158k	10281	154	1679	2011
P9-EDM	cloudbees-oss/zendesk-java-client	#480	15k	953	62	425	2013
P10-EDM	decorators-squad/eo-yaml	#504	15k	944	20	240	2016
P11-N–	dependencytrack/dependency-track	–	314k	3946	94	963	2013
P12-PNC	digitalpebble/storm-crawler	#974	51k	1815	39	344	2013
P13-N–	dius/java-faker	–	62k	834	83	515	2011
P14-PDM	eclipse/lemminx	#1228	511k	1305	38	824	2018
P15-PNM	ff4j/ff4j	#571	71k	1413	80	367	2013
P16-N–	firebase/firebase-admin-java	–	85k	447	42	610	2017
P17-EDC	gitlab4j/gitlab4j-api	#852	50k	2169	145	362	2014
P18-ENC	glyptodon/guacamole-client	#470	118k	6608	79	471	2013
P19-N–	hangarmc/hangar	–	66k	2874	41	860	2020
P20-N–	hibernate/hibernate-tools	–	51k	3177	16	4415	2011
P21-EDM	hyperledger/fabric-chaincode-java	#244	17k	490	35	282	2017
P22-N–	jenkinsci/email-ext-plugin	–	21k	1748	95	484	2010
P23-N–	jenkinsci/jira-plugin	–	15k	1481	79	546	2010
P24-PNC	jqno/equalsverifier	#654	36k	2884	31	542	2015
P25-ENM	jsqlparser/jsqlparser	#1568	52k	2030	112	420	2011
P26-ENO	jtablesaw/tablesaw	#1124	1.182k	2514	80	467	2016
P27-ENM	lukas-krecan/jsonunit	#530	14k	1549	39	461	2012
P28-PNO	maven-nar/nar-maven-plugin	#389	42k	1277	71	213	2009
P29-N–	mcmmo-dev/mcmmo	–	56k	6627	165	631	2012
P30-EDM	miso-lims/miso-lims	#2680	342k	4801	20	2596	2012
P31-PDC	moquette-io/moquette	#680	20k	1394	41	316	2014
P32-N–	mybatis/guice	–	16k	1809	25	520	2013
P33-ENM	nats-io/nats.java	#663	56k	1578	48	591	2015
P34-ENO	netflix/zuul	#1265	26k	1512	54	1080	2013
P35-PDC	nlpchina/elasticsearch-sql	#1179	145k	1010	30	250	2014
P36-PDM	oblac/jodd	#788	36k	5364	57	267	2012
P37-N–	open-metadata/openmetadata	–	639k	7322	176	7347	2021
P38-ENC	openhft/chronicle-queue	#1115	41k	7516	58	705	2013
P39-PND	perwendel/spark	#1257	12k	1067	124	528	2011
P40-N–	pwm-project/pwm	–	186k	3063	41	293	2015
P41-EDO	qos-ch/logback	#574	74k	4451	113	644	2009
P42-PDM	redis/jedis	#3019	70k	2269	188	1680	2010
P43-N–	redouane59/twittered	–	47k	701	24	278	2020
P44-EDO	rickfast/consul-client	#461	11k	556	72	255	2014
P45-PDM	rubenlagus/telegrambots	#1070	33k	1050	91	474	2016
P46-EDO	spotify/dbeam	#486	6k	821	14	645	2017
P47-ENC	spring-projects/spring-data-couchbase	#1461	40k	1210	48	589	2013
P48-EDM	synthetichealth/synthea	#1082	1.015k	4662	68	728	2016
P49-PDC	teamnewpipe/newpipeextractor	#850	155k	2479	64	642	2017
P50-PDM	wikidata/wikidata-toolkit	#691	44k	1891	28	553	2014
P51-PDC	xerial/sqlite-jdbc	#741	30k	1521	110	383	2014
P52-EDM	zsmartsystems/com.zsmartsystems.zigbee	#1333	165k	1180	29	1080	2017

over existing tests. For example, the method calls leading to the additional coverage were in code taken over from the original test, that was not influenced by the amplified change (P22-N–, P29-N–, P32-N–). In three other cases, we browsed through the existing tests for the same object and found tests that are already calling the instructions the amplified test claims to newly cover (P13-N–, P34-ENO, P43-N–). We found that in three false positive cases mocking was used (P13-N–, P29-N–, P32-N–), pointing to missing support for mocks in DSpot's coverage calculation.

Simple Getters and Setters with Non-Standard Names (P11-N–): Tests only contribute coverage in simple getters and setters with non-standard names (not starting with 'get' or 'set'), which should have been filtered by the prettifier.

Could Not Find Class (P37-N–, P40-N–): We could not find the test class and the class under test (P37-N–), or the class with additional coverage (P40-N–).

Test Did Not Pass (P2-EDC): A test did not pass because the expected exception was not thrown. This and the last issue could be caused by the time difference between the commit at which we amplified the tests and the commit on which our pull request was based.

Assertion Unrelated to New Coverage (P20-N–, P22-N–, P29-N–, P32-N–)**:** In four projects, we found tests where the generated assertion does not check the behavior of the newly covered code. For example, the assertion is generated at a location before the call to the newly covered code (P20-N–), or the checked value is not influenced by the newly covered code (P23-N–, P42-PDM, P48-EDM). In both cases, while the test *covers* the code, we cannot claim that it *tests* the code.

No Explicitly Thrown Exception (P17-EDC, P19-N–, P23-N–, P24-PNC): In four projects, we found tests for `RuntimeException`s implicitly caused, e.g., in an unprotected call on a parameter that was set to null during amplification. As these exceptions did not seem to be part of the developer-intended behavior, we excluded these tests.

Change Unrelated to Assertion or New Coverage (P6-N–):
We excluded tests where the amplified change did not influence the asserted value nor the additional coverage. The amplification process should check whether the amplified change is necessary for the additional coverage an amplified test is providing.

Readability and Understandability (P6-N–, P23-N–, P25-ENM, P38-ENC): A further negative selection criterion we used in four projects was that tests were not good to understand or not readable, because parts of them were cryptic, long, or verbose. For example, in P23-N– the original tests already contained complex configuration of mock behavior.

Unclear Connection between Test and Additional Coverage (P13-N–, P16-N–, P20-N–): In three projects, we encountered tests where it was unclear how the amplified change or the generated assertion leads to the new coverage reported by the amplification. In contrast to the coverage false positives, we did not find a test executing the same instructions, but we could not trace how the method calls in the new test would lead to execute the covered instructions.

Positive Selection Criteria

The positive selection criteria are divided into two groups: selecting the most valuable test, or one that we were curious about for our study. In seven projects, we did not need to apply any positive criteria, as there was only one test generated (P3-PDC, P9-EDM, P18-ENC, P21-EDM, P44-EDO, P47-ENC, P49-PDC). In 13 projects, the negative selection criteria

already excluded all generated tests, we excluded these projects from the rest of our study (P6-N–, P11-N–, P13-N–, P16-N–, P19-N–, P20-N–, P22-N–, P23-N–, P29-N–, P32-N–, P37-N–, P40-N–, P43-N–).

We used the following criteria for the positive selection:

Most Additional Coverage (P3-PDC, P9-EDM, P18-ENC, P21-EDM, P44-EDO, P47-ENC, P49-PDC): In six projects, the test we selected covered the most additional instructions. This takes little effort, as the tests in each class are already sorted according to their additional coverage contribution.

Understandability (P12-PNC, P15-PNM, P25-ENM, P26-ENO, P28-PNO, P39-PND, P41-EDO, P46-EDO): In nine projects, we selected tests based on their understandability, as we expect an easy to understand test to more likely be accepted. For this, three criteria emerged that we used in conjunction: a) the coverage improvement is local to a few, closely related methods, b) the connection from the test to the additionally covered methods is clear from the methods called in the test, and c) the test is small and simple.

On several occasions, we choose a candidate test because we were ***curious about the developer's reaction***. In all these cases, we still only considered tests we believe to be a valuable contribution to the project. Non-valuable tests are identified by the negative selection criteria discussed before.

Exception Test (P10-EDM, P17-EDC, P24-PNC, P30-EDM, P34-ENO, P35-PDC, P38-ENC, P42-PDM, P51-PDC): In nine projects, we selected a test that checks for an exception.

Could Be Considered Not Worth Testing (P7-EDM, P8-PDM, P31-PDC, P36-PDM, P45-PDM, P52-EDM): In six projects, the test was contributing coverage in methods that developers could consider not valuable to test, such as a complex setters, `toString`, or `equals`.

Documentation Mismatch (P27-ENM): In P27-ENM we selected a test whose behavior did not match with the documentation of the method under test.

Improve Assertion Manually (P33-ENM): For P33-ENM, we were curious if we can improve an assertion that is not checking the newly covered code.

Uncommonly Large Coverage Increase (P50-PDM): In P50-PDM, one small method call lead to a lot of new coverage, more than what we saw throughout the study.

> **Answer to RQ1.1:** When selecting tests for the pull requests, we mainly excluded coverage false positives, tests with assertions that do not check the newly covered code, or tests that check for unintended runtime exceptions.

4.4.3 RQ1.2: Which manual edits do we perform to improve the tests before submission?

In this section we present the checklist that we created to guide our manual editing step before opening pull requests.

Align Assertion Style (P4-EDM, P5-ENM, P7-EDM, P9-EDM, P10-EDM, P17-EDC, P25-ENM, P26-ENO, P27-ENM, P30-EDM, P33-ENM, P34-ENO, P38-ENC, P41-EDO, P44-EDO, P47-ENC, P52-EDM): The edit we performed in the largest number of projects (17) was to align the assertion style with the other tests. Examples include: statically importing `assertEquals`, and unifying the assertion framework, e.g., transforming plain JUnit asser-

tions to their Hamcrest versions. DSpot did not remove Hamcrest assertions, so we had to remove old, no longer matching assertions.

Remove Unnecessary Code (P2-EDC, P4-EDM, P5-ENM, P7-EDM, P10-EDM, P26-ENO, P34-ENO, P38-ENC, P41-EDO, P44-EDO, P47-ENC, P48-EDM, P52-EDM): The second most prevalent edit (13 projects) was to remove variables and statements that were not relevant for the asserted behavior of the amplified test. These are left over from the original test, or temporary variables created by the test amplification and missed during their intended removal. In rare cases we also had to remove unnecessary casts or parentheses, introduced by the test generation for safety.

Adapt To Match Other Edits (P5-ENM, P7-EDM, P18-ENC, P21-EDM, P33-ENM): In five projects, we had to adapt the description of the test to match our manual edits. In P5-ENM we also adapted the test name and the expected value of the assertion to match the behavior that changed during our edits.

Apply IDE Recommendation (P2-EDC, P17-EDC, P52-EDM): In three projects, IntelliJ proposed a simplification through static analysis, e.g., reducing an always true condition.

Resolve Formatting and Linters (P8-PDM, P10-EDM, P26-ENO, P46-EDO): The contribution guidelines of projects sometimes state to apply auto-formatting (P8-PDM, P10-EDM, P46-EDO) or resolve all linter warnings (P10-EDM) before finalizing a pull request. In P26-ENO, we added line breaks to long lines to improve the readability.

Change Test Name (P25-ENM, P34-ENO, P52-EDM): We changed the test name to avoid duplication with existing tests (P25-ENM, P52-EDM), or make the test name fit the convention of the other test names in the class (P34-ENO).

Resolve Unrelated Amplified Change, Additional Coverage or Generated Assertion (P5-ENM, P21-EDM, P33-ENM): We encountered tests where the amplified change, additional coverage, or generated assertion were unrelated. In two cases, we changed the assertion to check the behavior of the newly covered code (P21-EDM, P33-ENM). In P5-ENM and P33-ENM the amplified change and the new assertion provided additional coverage, but they were not related to each other. We selected one test goal and adapted the rest of the test.

Move Test (P1-EDC, P7-EDM, P10-EDM, P52-EDM): In two cases (P1-EDC, P10-EDM), the object under test and the additional coverage were not related to the test class of the original test. We moved the tests to a better fitting class. In two other projects (P7-EDM, P52-EDM), we added our tests below other tests that were targeting the same method.

Simplify Literals (P7-EDM, P21-EDM, P46-EDO): For three tests, we simplified literal values in the test setup. For example, we removed extra clauses from a constructed SQL query that were not relevant for the new test (P46-EDO).

Make Compile (P17-EDC, P25-ENM): In two projects, we found parameters that no longer fit the signature of the called method. We adapted them, e.g., by copying over variable initializations from other tests (P25-ENM).

Answer to RQ1.2: When manually editing the amplified tests, we most often aligned the assertions' style to the test class and removed code unnecessary for the test scenario.

4.4.4 RQ2.1: Which changes are proposed during the pull request discussion?

Here we present the changes discussed by the maintainers on the pull requests with amplified tests, structured along the categories that emerged from our analysis.

Code Style Conventions (P1-EDC, P8-PDM, P14-PDM, P33-ENM, P42-PDM, P47-ENC, P50-PDM, P51-PDC): Most frequently, the maintainers proposed changes to let the code adhere to style conventions [138–141]. This regarded aligning the static import of assertion methods (P1-EDC, P8-PDM, P14-PDM, P50-PDM) or used constants (P42-PDM) to the rest of the class, adding a blank line at the end of the file (P33-ENM), or listing our name among the authors of the file in the comment block (P47-ENC), resolving linter warnings (P1-EDC) to make the CI pass (P51-PDC), or adhering to variable naming conventions (P1-EDC). While these seem like conventions of the project, they were not explicitly stated in the contribution guidelines we examined before each pull request.

Remove Unnecessary Code (P12-PNC, P14-PDM, P49-PDC, P50-PDM): The next most frequently discussed change was removing unnecessary code. Three maintainers pointed to unused variables (P12-PNC, P49-PDC, P50-PDM). The test in P14-PDM saved the return value of a relevant method call in an unused variable. In P12-PNC the maintainer criticized a statement that had no impact on the test result, and in P49-PDC the reviewer pointed to unnecessary parentheses.

Change Test Name (P4-EDM, P8-PDM, P10-EDM, P14-PDM): In four pull requests the reviewers suggested changing the test name. The proposed names described the scenario of the method calls in the test (P4-EDM, P8-PDM, P10-EDM), or the exception expected by the assertion (P14-PDM). For P10-EDM, the maintainer explained their naming convention: *"all test names should follow the pattern xDoesSomething"*.

Practice Defensive Programming (P1-EDC, P4-EDM, P49-PDC): Over three projects we got five proposals related to defensive programming. The maintainer of P49-PDC suggested to not check for the complete message of an exception, which could fail if the code under test is refactored. The same reviewer asked to use interfaces instead of concrete implementations and to set variables as final where possible. The review of P4-EDM proposed to assert the return value of an intermediate call. The reviewer of P1-EDC advised to use the specialized try-with-resources when writing to an InputStream within a `try` environment.

Simplify Setup (P1-EDC, P8-PDM, P10-EDM, P14-PDM): The maintainers of four projects proposed to simplify the test setup. For example, in P8-PDM, we replaced a multiple times modified object with a fitting default instance. The reviewer of P14-PDM recognized that another call than one under test could throw the expected exception and proposed a change to avoid the tests passing because of the earlier thrown exception.

Choose More Powerful Assertion (P8-PDM, P10-EDM, P49-PDC): Three maintainers pointed to the benefit of using a stronger assertion method. For example, in P8-PDM they endorsed a change from `assertFalse(...equals())` to `assertNotEquals(..)`.

Merge or Extend Test (P3-PDC, P42-PDM, P48-EDM): Three projects discussed merging the contributed test with other tests for the same method. P3-PDC and P42-PDM pointed to moving the assertion to an existing test. The maintainer for P48-EDM proposed to add an assertion to test a second scenario in the method under test and was open to keep both in the same test or split them up into two unit tests.

Use Meaningful Scenario (P7-EDM, P25-ENM, P47-ENC): Three maintainers proposed changing the test setup to a more meaningful scenario. For example, the test for P25-ENM used default initializations for SQL queries. The reviewer of P25-ENM criticized that the queries were not meaningful, and asked to *"craft an actual valid expression."*
Move Test (P12-PNC, P50-PDM, P47-ENC): Three reviews asked to move the test to another class as it tested a different object than the original test modified by the amplification.
Change Assertion Message (P42-PDM, P30-EDM): The maintainers of P42-PDM and P30-EDM both proposed to change the assertion message to explain why the code throws the exception that is expected by the test case.
Move Test Data (P1-EDC): For P1-EDC we moved the amplified test to another class, including globally defined test data. The maintainer asked us to move the test data into the test itself, as it was the only test using the data.
Test All Scenarios (P48-EDM): In P48-EDM the reviewer proposed to add a second assertion, to let the resulting test check for both the succeeding and failing scenario.

Answer to RQ2.1: The majority of changes proposed during the pull request reviews were focused on adhering to code style conventions and removing unnecessary code.

Figure 4.6 looks closer at the connection between whether we manually edited a test and whether changes were proposed during the review. We observe that for both edited and not edited tests the maintainers were **more often proposing changes than not**. Three tests without edits were merged without any further changes, while in six projects the pull requests were closed even when changes were discussed. The latter happened, e.g., because through the discussion it became clear that the test is redundant to existing tests (P1-EDC), or the maintainers provided feedback on the code even though they already concluded to not accept the test (P49-PDC).

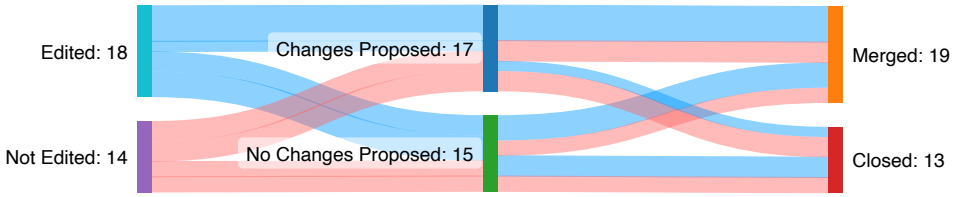


Figure 4.6: Flow of editing tests, changes proposed during the pull request and pull request outcome.

4.4.5 RQ2.2: What kind of information is requested by the maintainers during the pull request discussion?

Next to proposing changes, the maintainers also requested different kinds of information during the discussions:
Purpose of the Pull Request / Test (P3-PDC, P12-PNC, P25-ENM, P27-ENM): Four reviewers asked to explain the purpose of the pull request or the test, such as *"I'm unsure what issue this is targeting at resolving"* (P3-PDC), or *"what problem exactly will this PR solve?"* (P25-ENM).

Added Value (P2-EDC, P25-ENM, P27-ENM, P51-PDC): In four cases, we were asked about the added value that the test is providing.
Coverage Increase (P1-EDC): One maintainer included a coverage tool, checking the coverage increased.
Description about the Test (P33-ENM): For P33-ENM we did not include the textual description at first, but we were asked to add a description about our test into the pull request.
Contribution Compared to Existing Tests (P1-EDC): The maintainer of P1-EDC asked what our test checks in comparison to existing tests for the same method.
Curiosity (P7-EDM, P14-PDM, P24-PNC, P50-PDM): Three reviewers asked questions out of curiosity, such as "how [our tool] generated the parameter input" (P7-EDM), which IDE and formatter we used (P14-PDM), and how we came to writing a test for this specific method (P24-PNC, P50-PDM).

Figure 4.7 presents a closer analysis of the relationship between whether we provided a description in the initial pull request (such as in Figure 4.5) and whether additional information was requested by the reviewers (excluding curious questions). We can see that **questions appeared just as often whether we provided the generated description or not** (4 projects each), and two pull requests without description were merged without requests for more information. In contrast, **curious questions** on the details of our process **were mainly asked for pull requests with a description**. When we provided a description, giving additional information never lead to a merged pull request (4 projects), while the majority of pull requests with a description were merged without further requests for clarification (14 projects).

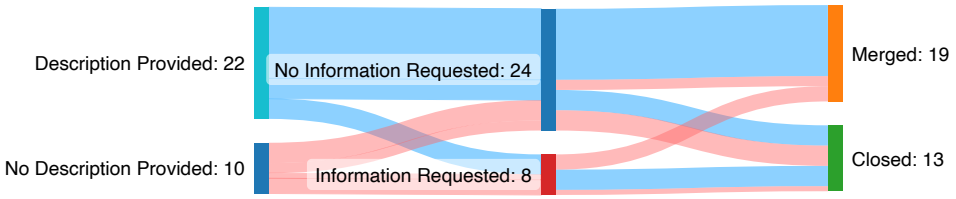


Figure 4.7: Flow of description provided, information requested during the pull request and pull request outcome.

Answer to RQ2.2: The maintainers mostly asked for more information regarding the purpose and value of the contributed test.

4.4.6 RQ2.3: How do the maintainers justify their judgment over the amplified tests during the pull request discussion?

Another aspect we analyzed were the reasons that reviewers accepted or rejected our pull requests.
Completeness of Contribution (P3-PDC, P5-ENM, P31-PDC, P47-ENC, P49-PDC): Three reviews pointed out that the contribution was not complete enough. This was because

all possible outcomes of a method should be tested (P31-PDC, P49-PDC), only a more comprehensive set of changes would be worth merging (P3-PDC), or an issue tracker entry (P5-ENM) needs to exist, and a discussion should happen before including a patch (P3-PDC).

Would Not Test (P2-EDC, P9-EDM, P49-PDC, P51-PDC): Three maintainers pointed out that the test was targeting methods they would not test, such as simple methods (P2-EDC, P49-PDC), classes taken from libraries (P51-PDC), or `toString` as it is used for debugging only (P9-EDM).

Test Untested Scenarios (P1-EDC, P9-EDM, P24-PNC, P27-ENM, P52-EDM): It was important to the reviewers that the proposed tests were testing yet untested scenarios. In P52-EDM and P9-EDM this was the rationale to merge the pull request, in P1-EDC and P24-PNC this was the reason to close the pull requests as the maintainers found other tests for the same scenarios. The reviewer of P27-ENM pointed out that *"ideally there should be some intention behind each test."*

Clear Test Scenario (P25-ENM, P27-ENM, P38-ENC): Three maintainers mentioned a meaningful scenario (P25-ENM) and clarity about what the test is testing (P27-ENM, P38-ENC).

Code Quality (P1-EDC, P31-PDC:) The reviewer of P1-EDC pointed out that the code should pass the linter. The maintainer of P31-PDC criticized that some code in the test is irrelevant for the method under test.

In several cases, we have no indication of the rationale for accepting or rejecting the pull request: Four projects merged (P15-PNM, P21-EDM, P36-PDM, P45-PDM) and two closed (P35-PDC, P17-EDC) our pull request without any comment.

Answer to RQ2.3: When verbalizing a rationale for their judgment on the amplified tests, the project maintainers mentioned the need for a comprehensive contribution of tests for meaningful, untested scenarios.

4.5 Discussion

In the previous section, we reported on the selection and manual edits we conducted before submitting the tests in pull requests, as well as the reactions of the maintainers concerning proposed changes, requested information, and rationale for their decisions to accept or reject the proposed tests. To connect our observations, we summarize the guidelines for developers to select and edit amplified tests in Table 4.2. Further in this section, we discuss the implications of our findings for developers that consider using developer-centric test amplification, and for test amplification researchers and tool designers. We also present threats to the validity of our study.

4.5.1 Guidelines for Developers to Select and Edit Amplified Tests

A strong take-away from our study is that the tests created by state-of-the-art test amplification tools still needed selection and editing efforts before they are incorporated into a maintained test suite. To summarize and connect the observations we made for our

Concern in Amplified Test	Connected Codes / Observations (Source RQ)	Explanation
Valid Coverage Improvement	Test Untested Scenarios (2.3) Added Value, Coverage Increase, Contribution Compared to Existing Tests (2.2) Coverage False Positive (1.1)	Check that the targeted code is not tested by another test (which might not be considered by amplification tool or coverage data)
Tests Relevant Code/Scenario in Project	Use Meaningful Scenario (2.1) Would Not Test (2.3) No Explicitly Thrown Exception (1.1)	Check that the new coverage provided by the test covers code that is relevant to test with your test suite
Only Necessary Code	Change Unrelated to Assertion or New Coverage (1.1) Remove Unnecessary Code (1.2, 2.1) Resolve Unrelated Amplified Change, Additional Coverage or Generated Assertion (1.2)	Check that all code in the test is relevant for the test's execution or understandability
Checks Behavior of Newly Covered Code	Assertion Unrelated to New Coverage (1.1) Resolve Unrelated Amplified Change, Additional Coverage or Generated Assertion (1.2)	Check that the assertion of the test actually validates the behavior of the additionally covered code
Test Scenario and Impact are Understandable	Readability and Understandability (1.1) Simplify Literals (1.2) Simplify Setup (2.1) Change Assertion Message (2.1) Unclear Connection between Test and Additional Coverage (1.1) Clear Test Scenario (2.3) Change Test Name (1.2, 2.1)	Check that you can / your colleagues could understand the test and what it is testing
Good Code Style, Adhering to Guidelines	Code Style Conventions (2.1) Align Assertion Style (1.2) Apply IDE Recommendation (1.2) Resolve Formatting and Linters (1.2) Change Test Name (1.2, 2.1) Practice Defensive Programming (2.1) Choose More Powerful Assertion (2.1) Code Quality (2.3)	Check that the code is well written and adheres to your guidelines
Appropriate Scope and Location	Move Test (1.2, 2.1) Merge or Extend Test (2.1) Change Test Name (1.2, 2.1) Move Test Data (2.1) Test All Scenarios (2.1)	Check that the test is at an appropriate location and has the right granularity (move/merge/extend with other test otherwise)

Table 4.2: Guidelines to select and edit amplified tests

five research questions, we present guidelines for developers on what aspects they should consider when reviewing an amplified test. Here, selection and editing are put together and the decision which action to take is left to the developer. If an issue is too large, or it it is unclear how to resolve it, the developer might choose to exclude the test entirely. If they see an easy change to address the issue, they might choose to edit the test and include it in their maintained test suite. Table 4.2 gives an overview and explanation of each of our guidelines, as well as the observations from our study that it is based on.

We recommend, that a developer using developer-centric test amplification, should review each test individually and consider whether:

- the newly covered code is indeed not yet covered by any other test,
- the newly covered code or scenario is relevant to be tested in their maintained test suite,
- the test only contains code necessary for its behavior or understandability,
- the assertion in the test validates the behavior of the newly covered code,
- the test behavior and its impact on the test suite is understandable to them and their colleagues,
- the code style is adequate and adheres to their coding guidelines,
- the test is at an appropriate location and whether it should be merged or extended with another test.

4.5.2 Relation to Existing Literature

Several of the edits to amplified tests we observed in our study are related to existing knowledge about high-quality tests and shortcomings in automatically generated tests. This section illustrates how each of our guidelines is supported by existing literature. However, to our knowledge, there is no research looking at what changes developers concretely make to generated or amplified tests before including them in a test suite.

Valid Coverage Improvement

Our first guideline is that the targeted code should not be covered by another test that might not have been considered by the coverage data used by the test amplification process. We observed something similar in an industrial study where developers considered code that was accounted for in other quality assurance practices or test suites to be not as relevant to test with a regression test (Chapter 6). In the concrete cases, the code blocks were covered by fuzzing, so the developers might have seen this robustness testing as sufficient. While improving an engineering goal such as coverage or mutation score is at the heart of the definition of test amplification [42], we see in this study that in practice we cannot always rely on the coverage data that test amplification tools use. This data might exclude other tests, higher-level test suites or other quality assurance practices.

Tests Relevant Code/Scenario in Project

When testing software, developers need to decide which code is worth testing with automated tests [2]. In other studies we conducted, we observed that not all code is relevant for developers to cover with regression tests (Chapters 2 and 6). This is in line with the common recommendation to not aim for 100 % code coverage [142, 143]. In interviews with developers, Kochhar et al. found that the judgment what to test is subjective, as participants disagreed whether it is useful to test simple things [40]. There can also be behaviors of code that should not be tested. Galindo-Guiterrez et al. identified checking for `NullPointerException`s that are not explicitly thrown in the code under test as an undesirable behavior of EvoSuite-generated tests.

Only Necessary Code

Our third guideline recommends removing all code that is not necessary for the execution of the test. This code might be left over from the original test that was amplified or no longer needed after other changes to the test. Similarly, the test smell "General Fixture" [144] is based on unnecessary code in test setup methdos, and unnecessary code is also a problem in production code [145]. Panichella et al. [146] propose to use optimization heuristics like purification [147], carving [148] or slicing [149] to improve generated tests by focussing them on one, semantically coherent scenario.

Checks Behavior of Newly Covered Code

Our next guideline concerns the assertions of the amplified tests, which should check the behavior of the newly covered code. It is well known that structural coverage can give an indication whether a test suite is bad, but does not indicate error detection and prevention strength [40, 150, 151]. The ability to reveal faults in the targeted production code is a criterion in Grano et al.'s quality factors for unit tests [61]. A miss-match between the act and assert phase of a test was one of the quality issues Galindo-Gutierrez et al. detected in tests generated by EvoSuite [152]. To address these issues, we could employ more refined metrics to select the amplified tests, such as checked coverage [153], oracle adequacy [154], or mutation score [155]. However, one must way the trade-offs regarding runtime, because such stronger metrics are generally more expensive to compute (Chapter 5). For mutation score, limiting the mutants to relevant lines [156], i.e., the additionally covered lines, could be an option to speed up computation. On the other hand, Zhang et al. [150] found that human-written assertions are stronger at detecting seeded faults than assertions generated by the tool Randoop [14].

Test Scenario and Impact are Understandable

The understandability of tests, or lack thereof, is mentioned in several user-involving studies on automatic test generation [35, 45, 47]. Code reviewers are concerned with the understandability of test that are contributed [59]. The understandability of a test is impacted by test names [31–33], variable identifiers [29, 34, 157], meaningful comments or summaries [34, 35], and the test data [73, 92, 158–160]. Lin et al. showed that the quality of identifier names is low in manually written and especially automatically generated tests [157]. The concern with readability of generated tests is a central motivation for the development of language model based test generation approaches [19, 161]. However, it was also shown that the judgment how readable a test is differs per developer [160],

and that experience influences the test comprehension process [92]. Daka et al. observed that developer-given test names could contain abstract knowledge about the test intent or scenarios, which was not the case for their generated names that focused on covered methods and asserted values [32].

Good Code Style, Adhering to Guidelines

Our guideline to ensure that the amplified tests have a good code style and adhere to the coding guidelines of a project, can also be observed in more general code review practices that require consistency of code style [162, 163]. Specifically for assertions, Zamprogno et al. found that developers prefer assertion statements that are consistent with the code style of the test suite [164]. While explicit guidelines on how to contribute to open source projects are more and more common [165], these documents often do not sufficiently reflect the whole process [166, 167] and especially lack information about not automatically checkable guidelines [168].

Appropriate Scope and Location

The final guideline in our list is to ensure that an amplified test has an appropriate scope and is in the right location within the code base. A too large scope, i.e., too much tested in one method, can be the test smell "Eager Test" [144] or a sign of lacking semantic coherence [146]. It also can make the test long, which negatively impacts understandability [29, 40]. Existing literature recommends that test code should be well-modularized and structured [40, 158]. Duplication of test setups over multiple tests is an indication of code clones hindering the maintainability of test code [152], which can be the motivation to merge an amplified test with an existing test from the test suite.

4.5.3 Implications for Practitioners

In this chapter, we characterized the selection and editing steps developers are likely to conduct before incorporating amplified tests into their maintained test suite. For software developers and project managers, our results can be the **basis to take an informed decision on whether to adopt developer-centric test amplification**, by providing a realistic view on the kind of adjustments required by developers. We divide these efforts into two groups: (1) actions that could be automated by customizing the test amplification to a project, and (2) actions that highly benefit from the developer's comprehension.

To the first category, we count the coverage false positives, additional coverage in not-test-worthy methods, adhering to code style guidelines, and using defensive programming constructs. If a software developer applies test amplification out of the box, without any further customization, they would run into these issues, such as we did during our study. However, if the project would commit to a longer use and invest the time in configuring and customizing the amplification tool for their project, such efforts can potentially be automated.

The other set of efforts require the software developers to understand the amplified test—which they aim for already before accepting the test. These efforts are about changing the scenario of the test to be simpler or more meaningful, removing left over code, moving or merging the test, or adding a clearer test name. With these, the test becomes easier to understand, therefore easier to maintain and more helpful when trying to locate the fault

when the test fails. These changes have a large impact on the quality of the resulting test, addressing commonly observed shortcomings of automatically generated tests [29, 45, 47].

4.5.4 Implications for Researchers and Tool Designers

Previous user studies on test generation and amplification have shown that software developers find it important to understand the produced tests [45, 46], (Chapter 2). Understanding the tests was also necessary for us when selecting and editing the amplified tests, just as for the maintainers, who asked for additional clarification when the test or the pull request description were not clear enough. During this study, we elicited several adjustments to amplified tests that require an understanding of the behavior of the test. We conjecture, that such edits are much easier for developers to perform than for an automated tool. The next step for researchers would be to **investigate whether test amplification collaborating with the developer** for changes that require understanding **is an effective alternative to automating** them.

Because understanding is a prerequisite for the developer's manual edits, we conjecture that it is crucial for developer-centric test amplification tools to **provide the information that developers need to understand and modify the amplified tests**. As we saw in Section 4.4.5, the descriptions we generate are one component that contributes here, pointing to the amplified changes and the additionally provided coverage. However, throughout our study we experienced that further information support is necessary. For example, visually connecting the methods called in the test with the additional coverage could help developers understand how the amplified test provides this coverage (Chapter 3). Developers would also benefit from knowing which other tests cover the same method [35, 94], to determine the difference to these tests, or to validate if all scenarios of a method are tested. When we performed changes to the test scenario, we were at times not sure whether the coverage reported by the test amplification tool is still provided. We hypothesize that a **close integration of test amplification and manual editing** would let the developer verify their changes in the terms of the test amplification tool.

While we plead to leverage the developer's understanding and expertise to collaboratively produce valuable tests, our results also point to possible improvements of the automatic amplification process. During our selection we encountered tests where the generated assertion was not checking the behavior of the newly covered code. One could **apply local mutation analysis to verify that an assertion is really checking the additionally covered code**, similar to Ma et al.'s commit-aware mutation testing [156]. This means applying mutations only to the newly covered code and evaluating whether they cause the amplified test to fail. This approach would have a better performance than selecting amplified tests on mutation score directly, and we could still use the more widely understood instruction coverage when communicating the value of a test to the developer [169].

We encountered amplified tests that are based on complex, manually written tests whereas their tested scenario did not need this complexity. We propose to **improve test amplification by smartly selecting the original test** to modify, starting from simple tests and continuing to more complex ones. This way, the simple cases that can be tested through test amplification are caught with simple original tests, and the more complex original tests are only used if the amplification covers scenarios that need this complexity.

In the edits we conducted ourselves, as well as the ones proposed by maintainers, we moved tests to other classes, because the test target of the amplified test was no longer the same as the target of the original test. Clearly **identifying the target of an amplified test** would empower amplification tools to propose a better location for the produced test, and to communicate the intended impact of the amplified test clearer to the developer. From our observations, the tests were moved to test classes that are related to the additionally covered methods, or related to the methods directly called in the test.

4.5.5 Threats to Validity

There are several threats to the validity of our results:

Reliability of Results

To ensure the consistency and reliability of our qualitative analysis' findings, the first two authors revised the emergent codes throughout discussions until they reached a negotiated agreement [134]. We also employed constant comparison [136], whereby each interpretation and finding is compared with existing findings as it emerges from the data analysis to increase the construct validity. Especially for the manual selection and edits we conducted ourselves (adressing **RQ1**), the background of the researchers might have influenced which issues we identified in the amplified tests. Present are the threats of confirmation bias and experimenter bias, where our previous experience of issues with amplified tests leads to us overly focussing on these issues. Independent evaluators with a different background with regards to test generation might have identified other issues. Even when considering the presence of these biases, we deemed the manual selection and editing necessary to avoid antagonizing the open source maintainers by submitting tests that are clearly not ready to be merged. To mitigate the impact of our background, we carefully structured and documented our selection and editing process through the checklists that form the answers to **RQ1.1** and **RQ1.2** and invite other researchers and software engineering practitioners to replicate our study and compare their findings.

Construct Validity

The deficiencies we observed in the amplified tests are closely related to the current state of the test amplification tool DSpot. It is the state-of-the-art for test amplification in Java, and the archetypical implementation of test amplification that other tools are based on [123, 170, 171]. Still, the selection and edit efforts will change when the automation improves in the future. If efforts we observed are automated, developers might be willing to make new kinds of changes to improve the amplified tests. Because we manually selected the amplified tests to submit in pull requests and edited half of them to improve their quality before submitting, the results to **RQ2** do not directly reflect current amplified tests, but rather what test amplification might be capable of in the future. To mitigate this, we carefully document and report the selection and editing checklists we used in the answers to **RQ1** and pull our take-away recommendations on both our manual efforts and the maintainers' feedback in the pull request discussions.

Participant Bias

We did not reveal that the tests were at least partially automatically generated, and the maintainers' feedback might change if they were aware of this. The maintainers could also

face a social desirability bias, answering in a way that they expect us or their surroundings to prefer. To mitigate this we did not reveal our exact research questions to them, and conducted the study in their familiar environment of pull request discussions.

Internal Validity

In most of the pull request discussions the maintainers did not communicate their rationale for accepting or rejecting the pull request. We hypothesize that such a judgment is based on a plethora of factors, e.g., the code quality or the coverage contribution. As visible in Table 4.1, our study includes a diverse set of projects, whose individual size, contributors, or general interaction with pull requests, might influence the acceptance of a pull request. To mitigate the threat of inferring too much from the pull request outcome, we focused our analysis on the concrete discussion comments from the project maintainers.

Generalizability

The threat of internal generalizability concerns whether the sampled study objects are representative of our population of interest: open-source Java projects. We only considered projects where DSpot did not fail during execution, and produced amplified tests within 30 min. Other software projects might need a considerably higher up-front effort to adapt the test amplification tool before they can apply it and show a different set of deficiencies in the produced tests. Projects that need more than 30 min to build or use external tools that cannot be set up with DSpot's plain Maven or Gradle support, might show a unique set of selection criteria and change wishes from the developers. To mitigate this threat, we focus on providing an overview of the possible selection and change efforts that developers can encounter. While we specify how often each of them occurred in our study, we refrain from hypothesizing how likely they would appear in any project.

With respect to external generalizability and external validity, we acknowledge the need for replication studies with other programming languages, test frameworks or project settings. The feedback from maintainers of less active projects could differ, and industrial projects could have different requirements for automated tests.

4.6 Related Work

In another open source contribution study [13], Danglot et al. showed that DSpot is able to provide valuable additions to existing test suites by amplifying tests. They amplified 40 test classes of 10 projects and opened 19 pull requests of which 13 were accepted. Compared to their study, we focus on comprehensively documenting which kind of manual adjustments are necessary before submitting an amplified test, conduct our study on a larger number of repositories and pull requests, and present a detailed analysis of the feedback from the open source maintainers. We previously conducted an exploratory study evaluating an IntelliJ plugin to facilitate developer-centric test amplification from within the developer's IDE (Chapter 2). While we gathered a broad variety of feedback through interviews with developers, this work focuses on the concrete changes that maintainers and code owners would make to the amplified tests, independently of IDE tooling.

There have been several studies of search-based test generation with EvoSuite that involved users [45–47]. Our findings corroborate several results from these studies, such as the importance of readability for the developers [45, 47], that the quality of a test is

strongly connected to how easy it is to elicit its behavior [47], and a diversity of preferences for tests between different developers [47]. Daka et al. [29] established identifiers, line length and constructor and method calls as important features of the readability of a test. We go further into analyzing what a developer would change to obtain a satisfactory test from a, potentially less readable, generated one. Similar to us, Almasi et al. [45] asked the participants of their industrial case study what they would change in the generated tests to keep them. Our findings corroborate their results that developers would change the test data, or scenario, and the assertions to more meaningful ones. In contrast to their study, our open source contribution study spreads over a larger variety of projects. We point to a greater diversity of concrete changes that were important to the projects we contributed to, such as aligning with code style conventions, or moving and merging tests.

In a large-scale, manual study of EvoSuite generated tests, Galindo-Gutierrez et al. [152] identified 13 new quality issues in automatically generated test cases, which are not covered by the previous definitions of test smells [144, 146, 172]. While our study is based on a different test generation approach and tool, several of their quality issues coincide with the deficiencies we observe in DSpot amplified tests, and which we recommend developers to consider when selecting and editing amplified tests. They name three quality issues concerning a mismatch between the act and assert sections of the test case, which correspond to our observations of unrelated amplified change, additional coverage or generated assertion (**RQ1**). Our filter criterion "No Explicitly Thrown Exception" is also present in their list of quality issues. A set of their collected issues does not apply to the approach of test amplification, where one test is generated and then integrated into an existing test suite. These issues concern code and test scenarios that are redundant between the many tests EvoSuite generates, or violate the stricter unit testing paradigm aimed at by EvoSuite, i.e., only testing behavior directly in the class under test.

Incorporating the developer's expertise into the test amplification process, is also central in interactive search-based test generation [78, 173, 174]. In contrast to this field, we do not ask the developer to provide specific types of judgments to improve the search process, but instead they customize the amplified test to its final state for their test suite.

Several previous works investigated generating descriptions for automatically generated [34, 35, 175] and manually written tests [72, 176] and have shown that these descriptions help developers understand the tests [34, 35, 72]. Similar to us, these approaches leverage the called and covered methods to describe the intention of the test case. Our description is specialized for amplified tests, focussing on the amplified change and new assertion, while referring to the original test, leading to a shorter description.

4.7 Conclusion and Future Work

In this chapter, we manually analyzed the amplified tests of 52 projects, and discussed them through 39 pull requests with their open source maintainers. In a nutshell, we contribute:

- Insights into the selection and manual editing we performed to prepare the amplified tests for a pull request.
- Insights into the proposed changes, requested information and judgment of open source maintainers towards developer-centric amplified tests.

- Improvements to the test suites of 19 open source projects through our accepted pull requests.

Throughout the whole study we repeatedly observed that amplified tests need to be understood by developers before they consider including the tests into their maintained test suite. This understanding was also the basis for several kinds of edits we made and changes that were proposed by the maintainers, opening up a fundamental question for researchers working on developer-centric test amplification:

> Should we focus on further automating test amplification or focus on supporting developers in understanding the amplified tests, leaving some edits to them?

Therefore, the next steps in this line of research are to investigate this tradeoff and to develop tools that support developers with information and actionable recommendations while editing amplified tests. Further, we want to improve the state-of-the-art of test amplification by automating the now manual efforts and by sharpening the quality of the amplified tests through local mutation analysis. We encourage researchers to validate whether our results hold for other programming languages and test generation tools.

5

When to Let the Developer Guide: Trade-offs Between Open and Guided Test Amplification

Test amplification generates new tests by mutating existing, developer-written tests and keeping those tests that improve the coverage of the test suite. Current amplification tools focus on starting from a specific test and propose coverage improvements all over a software project, requiring considerable effort from the software engineer to understand and evaluate the different tests when deciding whether to include a test in the maintained test suite. In this chapter, we propose a novel approach that lets the developer take charge and guide the test amplification process towards a specific branch they would like to test in a control-flow graph visualization. We evaluate whether simple modifications to the automatic process that incorporate the guidance make the test amplification more effective at covering targeted branches. In a user study and semi-structured interviews we compare our user-guided test amplification approach to the state-of-the-art open test amplification approach. While our participants prefer the guided approach, we uncover several trade-offs that influence which approach is the better choice, largely depending on the use case of the developer.

This chapter is based on *C. Brandt, D. Wang and A. Zaidman. When to Let the Developer Guide: Trade-offs Between Open and Guided Test Amplification, IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2023* [55]. The design, implementation and experiments were conducted by Danyao Wang during her Masters Thesis under the supervision of Carolin Brandt and Andy Zaidman.

Software testing is one of the central activities in the software development lifecycle [58]. One part of this are developer tests, i.e., small automated programs that software developers write to check that their code behaves as they intend and prevent it from breaking in the future [3]. While developer testing is widely seen as valuable, it is also a tedious and time-consuming activity [8]. One automated approach to relief developers of this manual effort is *test amplification*. Test amplification mutates existing, developer-written tests to explore new behavior of the code under test [42]. Previous studies have shown that it can provide valuable tests to developers [13, 21], [Chapter 2], but at the cost of long runtimes [21], [Chapter 2] and effort for the developers to understand the behavior and impact of the amplified tests [36], [Chapters 2 and 3]. Let us illustrate this with an example:

> Masha, a software developer, is working on a new feature of their software project, that requires small changes in their existing code. Before submitting a patch, she needs tests that cover all her new code, so she decides to use test amplification to generate them automatically. She picks an existing test from the class she worked on and asks the tool to create new tests based on it. After a while the tool reports back and proposes several tests to her. Unfortunately, the class did not have a high test coverage, so she has to sift through quite a few tests spending time to understand what code they cover and realize it is not the code she is concerned with. Even for the tests that target her code, she has to switch between several methods under test and every time recall what behavior this method should have, so she can judge whether the generated test is correct.

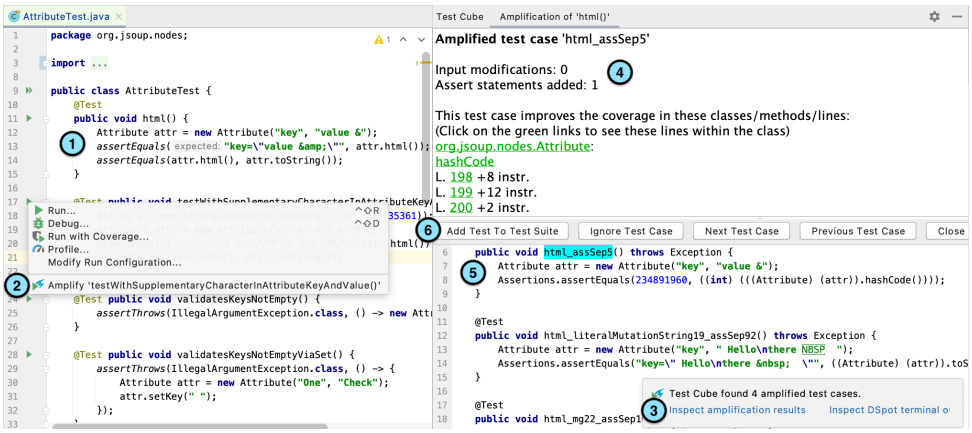


Figure 5.1: Interaction with our original test exploration IDE plugin for open test amplification [Chapter 2].

Our hypothesis is that these understandability issues are in part rooted in the disconnect between the present point of interest of a developer in the code base, and the dispersed coverage contributions amplified tests are providing, i.e., they need to rebuild the task context [177]. To bridge this disconnect, we propose to involve the software developer

more tightly in the test amplification process. Ideally, they can convey what piece of code they are interested in to test and then the test amplification presents only those tests that are relevant for the focus of the developer.

In this chapter, we propose a novel approach of user-guided test amplification. Starting from a method in their code base, the developer can initiate the test amplification and choose in a visualized control-flow graph which branch of the method should be tested. The test amplification is then directed to call this method specifically, and generates a variety of tests for it. It measures the tests' branch coverage and presents all tests that cover the intended branch to the developer, using the same control-flow graph visualization to help the developer understand how the test executes the method under test.

We conduct a technical case study and a user study to understand the impact and potential use of user-guided test amplification.[1] In both studies we compare it to the existing test amplification approach [13], [Chapter 2], which we will call *open test amplification* for a clearer distinction. With our technical case study on 31 classes from two open source projects, we investigate whether our simple changes in the guided amplification process are indeed effective at producing a higher ratio of tests for the targeted branch, and whether to guidance enables us to cover more branches overall in a project. Our findings from this study answer our first research question:

RQ1: How effective does guided test amplification generate tests for targeted branches (compared to open test amplification)?

In our user study, 12 developers apply both approaches to two classes and we interview them about their experiences. From this, we learn how they perceive each technique and their considerations when comparing them to each other. Our observations address our second research question:

RQ2: How do developers perceive guided test amplification (compared to open test amplification)?

Our two evaluation studies show that user-guided test amplification does deliver on the intended goals of making the test amplification process more effective and the coverage of the amplified tests easier to understand. However, the studies also show that the user-guided version of test amplification is not always better. From the participant's explanations during the interviews we learned that user-guided test amplification is closer to the real-life process of developing and testing new code where the developer focuses on a specific feature, writing code and tests for it. On the other hand, open test amplification is more suited when focusing on improving the test suite for an already existing code base, as it connects new tests clearer to the already existing tests. This is one example of the trade-offs between open and user-guided test amplification that our studies make apparent. We discuss all trade-offs we encountered to help the reader understand the strengths and

[1] We follow the empirical standard for engineering research: `https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=EngineeringResearch`

weaknesses of both approaches, and to help developers choose which approach fits best to their goals and workflow.

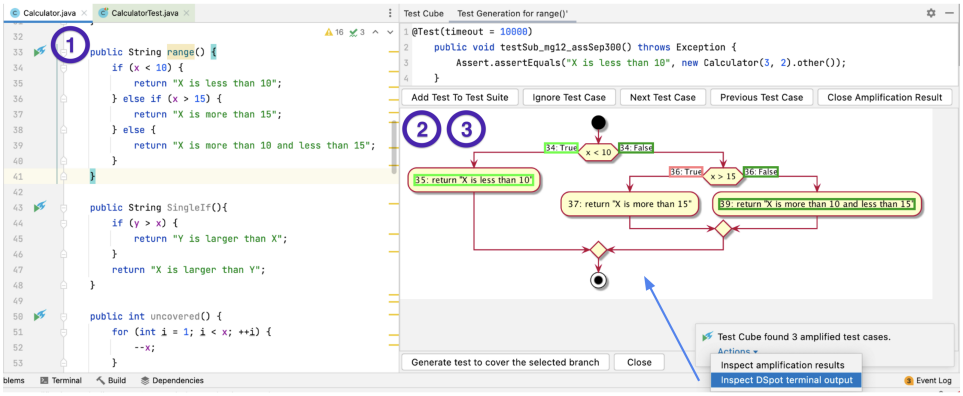


Figure 5.2: Interaction with user-guided test amplification.

5.1 Test Amplification

In this section, we introduce the concept of (open) test amplification, which is realized in the state-of-the-art test amplification tool for Java called DSpot [13].

The aim of *test amplification* is to generate new tests by leveraging the knowledge in existing, human-written tests [42]. These new tests improve the existing test suite with respect to a defined engineering goal, e.g., structural coverage or mutation score. Our work is based on our original design of *developer-centric test amplification*, which focuses on generating short and easy-to-understand tests to be included into the developer's maintained code base [Chapter 2].

A central part of developer-centric test amplification is to combine the automatic test amplification with a *test exploration tool* that guides the developer's interaction with the test amplification. Figure 5.1 illustrates the workflow with their prototype in form of an IDE plugin. The developer starts by selecting an original test to be the basis for the amplification ① and requesting the plugin to amplify that test ②. When the amplification finishes, it notifies the developer ③ that they can start exploring the generated tests. The exploration tool presents to the developer the additional coverage that an amplified test provides ④, the code of the test ⑤, and action buttons to easily add the test into the test suite or browse through the list of amplified tests ⑥.

The automated process behind test amplification (see the upper half of Figure 5.3) starts from an original test which comes from the existing, manually written test suite of a software project. We mutate the input phase of the test with several amplification operators: changing literal values slightly or replacing them with random values, as well as adding, duplicating or removing method calls to the objects under test. The old assertions are replaced by new ones which use the current behavior of the system as the oracle. Then, we execute all new tests and measure their instruction coverage. The tool selects all tests

that cover new instructions compared to the existing test suite and presents them to the developer.

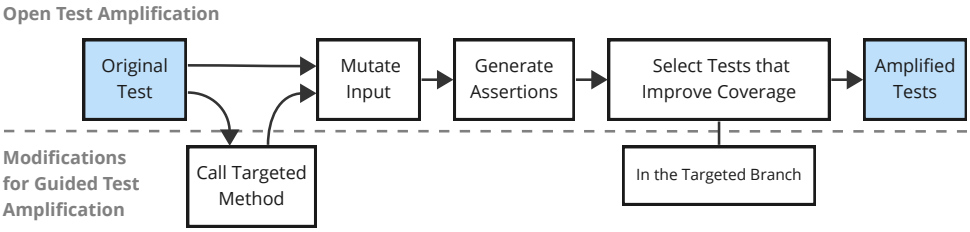


Figure 5.3: The automated process behind open test amplification and modifications to it for guided test amplification.

The interaction and the underlying amplification process starts from a developer-selected original test, randomly mutates it and keeps all new tests that cover new instructions anywhere in the project under test. We coin it *open test amplification* as it openly looks for any new tests that could be valuable for a project.

Previous studies on open test amplification showed that with this approach, it is difficult for the users to connect the test to the code under test it covers [Chapters 2 and 3]. Also, not all uncovered code is equally important to be tested in the opinion of the developers [Chapters 2, 4 and 6] When we originally proposed this approach, we had to take several design decisions that limit the power of the amplification, in order to make it fast enough to be interactively used [Chapter 2]. To address these shortcomings, we propose to let the developer take the lead and guide the test amplification towards the code they find relevant to be tested.

5.2 User-Guided Test Amplification

To speed up the process of finding new tests and make it easier for the developer to understand the context of the generated tests, we propose to let the developer direct the test amplification to the specific code they want to test. We call this approach *user-guided test amplification* and build it upon the developer-centric implementation of DSpot [13], [Chapter 2].

The developer starts by selecting a method in the code under test which they would like to test (see ① in Figure 5.2). Then, the test exploration tool presents them with a control flow graph of that method, similar to the graph shown at ②. The graph shows the execution structure of the method through boxes for each statement and condition, connected with arrows. The arrows annotated with "True" or "False" represent branches in the control flow of the method, letting the developer see the different scenarios that might need testing. We compute the existing test coverage for the method and highlight the branches that are already covered in green, and those that are not covered in red. The developer can select the branch that they would like to cover and start the test amplification. The tool automatically looks for the corresponding test class and picks the first—often most simple—test as the original test for the amplification. If no corresponing test class or test can be found, the tool prompts the user to create a test and invoke the amplification again.

When inspecting the result, the test exploration tool reuses the same control flow graph to show the developer the additional coverage that the amplified test provides ③. The developer can then decide whether to add the test to the test suite or to continue exploring the other tests or invoke the tool again for other branches.

We add two simple modifications to the underlying automated test amplification process to incorporate the guidance provided by the developer. The lower half of Figure 5.3 illustrates the modifications we make to the open test amplification process. As the first modification to the input of the original test, we call the method selected by the developer with randomly generated values for the parameters. When an object is needed, DSpot looks for a public constructor and uses it with random values to initialize the object. Then we continue by randomly mutating the test input as with open test amplification. All produced tests that cover the branch selected by the developer are selected as results to be presented to the developer.

We intentionally make simple modifications and largely rely on the amplification operators available in the base tool DSpot, e.g., the random generation of parameter values for object initialization. Our aim is to see whether such simple changes can already be effective to improve test amplification before considering more complex and runtime-impacting alternatives.

5.3 Evaluation

To evaluate our proposed user-guided test amplification, we conduct two comparative studies: a technical case study and a user study. Our first goal is to judge the effectiveness of our technical changes to the test amplification process: does the guidance lead to a larger proportion of the generated tests covering the targeted branch compared to using open test amplification (**RQ1**)? The second goal is to elicit the opinions of developers on interacting with user-guided and open test amplification (**RQ2**).

RQ1: How effective does guided test amplification generate tests for targeted branches (compared to open test amplification)?

RQ2: How do developers perceive guided test amplification (compared to open test amplification)?

To answer **RQ1** we conduct a technical case study, where we apply both approaches to generate tests for 100 branches sampled from 31 classes of two open source projects. We analyze the ratio of amplified tests fulfilling our coverage goals to determine which approach is more effective. To answer **RQ2** we perform a user study with 12 developers that apply both open and guided test amplification to test two classes. Then we interview each participant to elicit their impression of each approach and how they compare to each other.

5.3.1 Design Technical Case Study

In our technical case study, we sample code branches from two open source projects and apply both guided and open test amplification to try to cover them. We measure how many

branches can be covered at all by each approach, and what percentage of the amplified tests generated in one run cover the targeted branch.

We select two open source projects as study objects: Javapoet [2], a library to generate java source files, and Stream-lib [3], a library for summarizing data in streams. An important selection criterion was the traceability from code to tests: in both projects we can identify the matching test class for a class, because they adhere to consistent naming conventions. To select the targeted methods under test, we pick all classes with a clearly identified test class and from these classes select all public, non-static and non-abstract methods, which are the methods that can be called by DSpot's amplification operators. Taking all branches from the selected methods under test (160 from Javapoet, 264 from Stream-lib), we randomly sampled 100 branches per project. From their matching test class, we take the first test as the original test method for the amplification.

We run both guided and open test amplification for each of the sampled branches, limiting the number of produced tests to 200 per run. Next, we collect all resulting tests as well as their coverage information. Per project, we calculated the ratio of covered branches over the sampled branches (Equation (5.1)).

$$ratio\ covered\ branches = \frac{\#\ branches\ covered}{\#\ branches\ sampled} \tag{5.1}$$

We calculate for each approach per project the average ratio of successful tests (Equation (5.2)) over all runs. The ratio of successful tests looks at how many of the returned amplified tests do indeed cover the targeted branch.

$$ratio\ successful\ tests = \frac{\#\ tests\ covering\ branch}{\#\ tests\ returned} \tag{5.2}$$

5.3.2 Results Technical Case Study

Table 5.1 shows the calculated effectiveness of guided and open test amplification in comparison. We see that the guided test amplification can cover more branches in both projects, but the difference is small, and neither approach can cover more than 41% of the sampled branches. This shows that guiding the test amplification by explicitly calling the method that contains the targeted branch is only marginally helpful in covering a larger variety of branches of a project.

Table 5.1: Ratio of covered branches (see Equation (5.1))).

	Javapoet	Stream-lib
Open Test Amplification	23%	35%
Guided Test Amplification	32%	41%

To understand why many branches could not be covered by either test amplification approach, we manually inspected the branches that could not be covered. A core reason

[2] https://github.com/square/javapoet
[3] https://github.com/addthis/stream-lib

for not covering a branch was that the objects under test or the target method parameters are not initialized with the right values. In some cases, this came from the amplification tool not supporting the parameter's type, e.g., for a class without a public constructor. Then, the tool sets the parameters to null or empty values, which lead to exceptions when trying to generate assertions. We saw that Javapoet's classes have more methods whose parameter types are classes without public constructors, while Stream-lib mostly works with simple data types for the parameters. As the amplification tool's implementation does not support initializing classes without public constructors, this could explain why the amplification is more effective on Stream-lib than on Javapoet. Similarly, generating tests for faults or locations that require complex input objects is challenging for search-based tools like EvoSuite [45].

We investigated whether the choice of the original test impacts the ability to cover a certain branch. For this, we sampled ten branches that were not covered by the amplification and also not the existing test suites. Then, we amplified all tests in the corresponding test class, but still could not generate tests that cover the sampled branches. This shows that selecting different initial tests likely does not impact how effective the test amplification is at covering the sampled branches. The earlier mentioned likely cause for not covering the branches, not being able to generate the right initialization for the objects under test, seems to not be solved by selecting different initial tests.

Table 5.2 shows how many of the tests generated in one run of guided and open test amplification cover the targeted branch. While the ratio of tests that successfully cover the targeted branch with open test amplification is only 24% for Javapoet and 45% for Stream-lib, for guided test amplification this ratio is 70% for both projects. These results show that the guided test amplification is substantially more likely to produce tests that cover the targeted branch. This indicates, that the simple guidance we implemented into the guided test amplification—calling the method containing the targeted branch—is indeed effective at guiding the test amplification towards our target. Therefore, using guided test amplification enables us to set the amplification to generate fewer tests, while still having a good chance at receiving a test that covers the targeted branch.

Table 5.2: Average ratio of successful tests, which cover the targeted branch (see Equation (5.2)).

	Javapoet	Stream-lib
Open Test Amplification	24%	45%
Guided Test Amplification	70%	70%

Looking how the ratio of successful tests is distributed over the sampled, targeted branches (Figure 5.4), we see clear differences between the projects. While in Javapoet the distributions are dense and the higher effectiveness of guided test amplification is clearly visible, for the Stream-lib project the ratio of tests successfully covering the targeted branch differs much more significantly from branch to branch. One possible explanation for this difference is that number of methods in Javapoet's classes is higher than in Stream-lib. This means that it benefits more from the modification in guided test amplification that explicitly calls the method under test before the further input mutation.

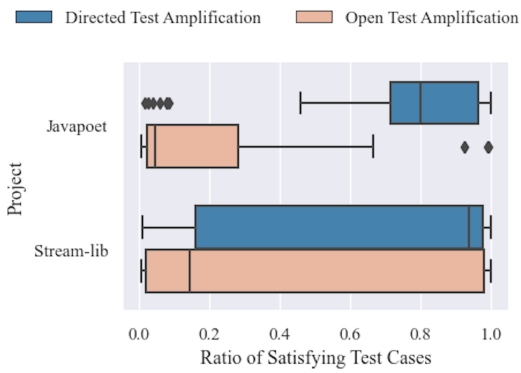


Figure 5.4: Distribution of the ratio of successful tests (see Equation (5.2)).

5.3.3 Answer to RQ1: How effective does guided test amplification generate tests for targeted branches (compared to open test amplification)?

Summarizing the results of our technical case study, we can see that **guided test amplification is more effective than open test amplification** when covering a specific targeted branch. However, both approaches fail to cover the majority of the sampled branches and depending on the project there can be a large variety in the ratio of generated tests covering the targeted for both approaches. We will discuss and interpret these observations together with the insights from our user study in Section 5.4.

5.3.4 Design User Study

Our central ideas for guided test amplification were motivated by the interaction with the user: the developer initiates the test amplification and guides it towards a method and branch, reducing the search space for new tests. In addition, this should help the developer understand and review the generated tests, because they already built up the necessary mental task context of the method unter test [177]. To elicit the opinions of developers on the use of guided test amplification in comparison to open test amplification, we conduct a study.

The user study starts with a questionnaire collecting demographic information and informed consent from each participant. Then, the participants are introduced to the concept of test amplification and asked to generate tests for two classes with similar complexity taken from the open source project Stream-lib. Each developer applied both open and guided test amplification, and we equally shuffled the order of the approaches and which class they test according to the four groups in Table 5.3. After the participant solved both tasks, we conduct a semi-structured interview. Guided by a list of closed questions (see Figures 5.5 and 5.6) we ask the participants to reflect on their experience with the open and guided test amplification, to compare both approaches and to express their overall impression of the amplified tests.

We conducted the study fully remotely in sessions of 60 min to 90 min. We recruited

12 participants through convenience sampling in our professional networks and on social media. You can find the complete tasks and questionnaires in our online appendix [178]. Our study design was approved by our local ethics review board.

Table 5.3: Task ordering for our participant groups.

Group	First Task	Second Task
1	User-Guided Test Amplification *StreamSummary*	Open Test Amplification *ConcurrentStreamSummary*
2	User-Guided Test Amplification *ConcurrentStreamSummary*	Open Test Amplification *StreamSummary*
3	Open Test Amplification *StreamSummary*	User-Guided Test Amplification *ConcurrentStreamSummary*
4	Open Test Amplification *ConcurrentStreamSummary*	User-Guided Test Amplification *StreamSummary*

5.3.5 Results User Study

From the demographic questionnaire, we learn that we have a relatively young population of 12 participants with a development experience of one to three years (7), four to six years (4) and seven to nine years (1). Two of the participants had used an automatic test generation tool before. Their main programming languages were Python (6), Java (4), or C++ (3), and they mainly identified as working in general software development (4), research (2) or data and analytics (2).

Guided Test Amplification

Looking at the feedback regarding the guided test amplification, presented in Figure 5.5, the participants strongly agree that the control flow graph showing the coverage of the target method is easy to understand (Q1). When asked whether the information provided is valuable, the participants strongly agree (Q2) and point out that the primary value is in visualizing the code structure and coverage, especially when the complexity of the method under test is high. Question (Q3) centers around whether the control flow graph effectively lets the participants convey their expectation of what to cover to the amplification. On average the participants agree to this, pointing out that it also helps identify all scenarios that are possible when calling the method under test.

They agree that the same visualization is also easy to understand when it comes to showing the coverage of an amplified test (Q4), and helps to select which amplified test to keep and add into the test suite (Q5). In this selection process, the visualization was especially helpful when the amplified tests provided diverse coverage contributions in methods with many branching points. Two participants were neutral about using the control flow graph to select a test, pointing to that they only want to cover the previously selected branch and rather focus on the code of the amplified test instead when selecting or add the test without further inspection.

Open Test Amplification

When it comes to the open test amplification, our study participants are more divided, but on average agree that the text-based instruction coverage explanation is easy to understand

(Q6, Q7) and provides useful information (Q8). The main complaints were that listing each occurrence of new instruction coverage was too detailed and that the connection between the test and the covered instructions was not clear even with the provided hyperlinks. The participants that were positive found the class and method names informative and liked that the hyperlinks let them locate the code under test conveniently. We asked whether the provided information about the amplification mutations in the test (Q9) and the additional coverage (Q10) helped the developers select which test to keep. The participants on average agreed that the additional coverage is helpful to select which test to keep (Q10). However, they criticized that they could not see the existing coverage to judge if a line in the code under test is already covered or not. One participant also thought out loud about whether the provided coverage is actually important coverage.

Both Approaches Compared

After discussing each amplification approach separately with our participants, we asked several questions to compare both approaches (see Figure 5.6). Directly asked whether the instruction coverage of open test amplification or the branch coverage of guided test amplification is easier to understand, all participants prefer the branch coverage (Q13). The participants found it easier to map the branch coverage to the source code structure. Some were also not familiar with the concept of instruction coverage and struggled to identify the single instructions in a line of code. Most participants prefer the visualized control flow graph over representing coverage as highlights in the editor (Q14). Using the visualization they did not need to read the source code of the method under test.

We asked the developers to reflect which approach helps them more during test generation (Q16) and they were divided between the two approaches. Seven participants prefer the guided test amplification as it is closer to writing tests in real-life scenarios, where they focus on specific features to cover. Two participants prefer open test amplification: one proposes to use it early in the test creation process to cover as much code as possible, the other focuses on connecting a new test with the existing ones it is based on, which is clearer during open test amplification. Three participants were neutral and voted to combine the two approaches. When they do not have a specific coverage goal they would use open test amplification, while they would choose the guided test amplification when they aim for more control over each tests' coverage.

Regarding selecting which resulting test to incorporate into the test suite, the participants mainly prefer the guided test amplification (Q15). The ten participants voting for guided test amplification mention that when writing tests they usually have a specific feature in the code they want to cover, which they can achieve by guiding the test amplification. One participant prefers open test amplification as they focus on covering the whole project as much as possible and want to compare the different tests based on their total contributed coverage. One participant is neutral and would use both approaches depending on the situation.

Finally, we asked about their overall impression of the amplified test, which was positive (Q11, Figure 5.5). The participants on average strongly agree that they would use test amplification again (Q12) and gave a variety of suggestions on how to improve the tools for both test amplification approaches. One aspect they noted positively is that the tool clearly indicates when it could not generate a test for a selected branch, which made these situations less negative in the participants' opinion.

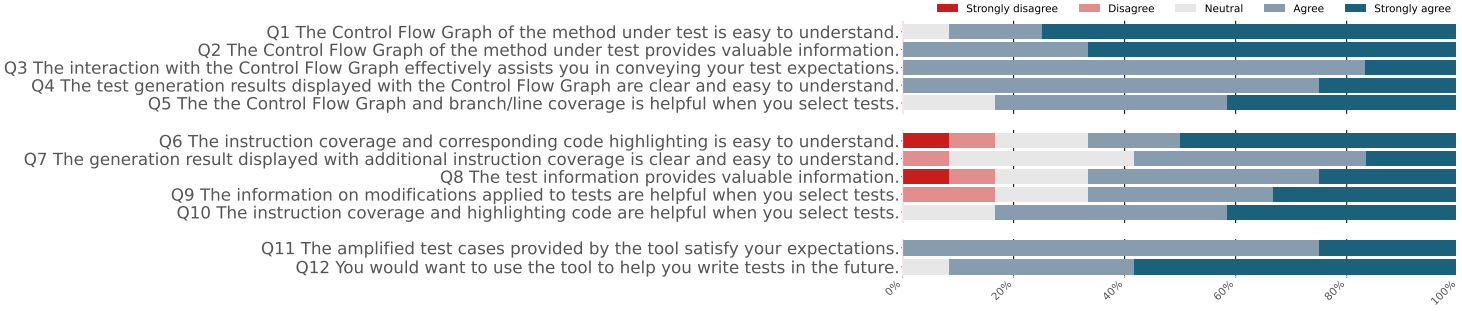


Figure 5.5: Participant answers on each of the two amplification approaches and test amplification in general.

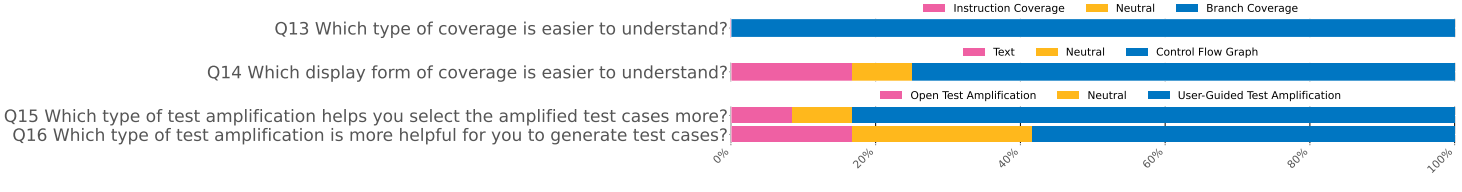


Figure 5.6: Participant answers on comparing user-guided and open test amplification.

5.3.6 Answer to RQ2: How do developers perceive guided test amplification (compared to open test amplification)?

Looking at all the results of our user study, we see that a majority of our **participants prefer the user-guided test amplification** approach (Q16) because it **fits better into the typical situation they create tests in**: when they want to test a specific location in their code. Factors contributing to this judgement are that all participants found branch coverage easier to understand than instruction coverage (Q13), and most preferred the structure-revealing control-flow graph visualization over the more precise textual representation of additional coverage (Q14). This preference for user-guided test amplification is also supported by the overall more positive ratings in the detailed questions about the approach (Q1-5), compared to the detailed questions about open test amplification (Q6-10).

From the explanations of our participants we learned that they do not universally prefer user-guided test amplification over open test amplification, but that it depends on their use case, the information that they need to judge the amplified tests and the amount of control they want to have over the amplification process. The results of our technical study showed that the effectiveness of guided test amplification compared to open test amplification depends on the class structure in the code under test and the data types used as parameters. Taken together, we see that there are **trade-offs between the two approaches** that should be considered when choosing either to work with or to improve in future research. In Section 5.4 we collect these trade-offs and discuss the implications of them for practitioners and researchers.

5.3.7 Threats to Validity

There are several threats to the validity of our two studies and their results. When it comes to *internal validity*, we mitigated the threats by switching the order of the two approaches (threat: learning effect) and which class each approach was applied to (threat: dissimilar classes) equally over the four randomly-assigned participant groups. The characteristics of the two projects and their classes in our technical study could dictate the outcome of our technical study. To mitigate this, we manually analyzed the classes and transparently discuss the impact of the number of methods per class and the complexity of the used data types on the effectiveness comparison of the test amplification approaches. To ensure the *confirmability* of our user study results, we focus on presenting the closed question ratings and support them with explanations staying as close as possible to the participants' formulations.

Regarding *construct validity*, the results of both studies are influenced by our prototype implementations. We used the same test amplification tool for both approaches, which is based on DSpot and limited to Java, with the only differences in implementation described in Section 5.2. Another threat is whether we are measuring the effect of the different amplification approaches or the changed user interface (UI) from open to user-guided test amplification. Just as in our original study on developer-centric test amplification [Chapter 2], we believe that a tool for developers and its UI can fundamentally not be developed or studied in isolation. To mitigate this threat, we ask separate questions about the information and the UI elements to our participants (Q1/2, Q4/5, Q6/10, Q13/14).

The *external validity* of the results from our technical study is threatened by the two projects selected for the case study. We observed that the complexity of the used data types

and the number of methods in a class influence the effectiveness of the test amplification. Further studies on a larger variety of projects and classes are needed to demonstrate the generalizability of our findings. Another threat to the external validity of our user study is whether the participants experienced the whole variety of methods which to test with amplification. To mitigate this, we selected example classes with a varied complexity of methods and initial tests that cover some methods of the class fully, partially or not at all. In the user study we have participants from a range of different software domains, but no participant has more than ten years of development experience, making the results potentially not generalizable to very senior developers.

5.4 Discussion and Implications for Practitioners and Researchers

With designing user-guided test amplification, we set out to improve the effectiveness of the process and the understandability of the produced tests. Our technical case study indicates that user-guided test amplification is indeed more effective, and the user study suggests that developers find its components more understandable than those of open test amplification. However, we also saw that the effectiveness of each approach varies per project and class, and that the developers might prefer different test amplification approaches depending on their current goal with testing. In this section, we will discuss a series of trade-offs that we identified based on our two studies and the design of both amplification techniques. Table 5.4 gives an overview of these trade-offs, together with the source from which we take the answer for either technique.

Table 5.4: Trade-offs between user-guided and open test amplification.

	User-Guided Test Amplification	Open Test Amplification
Fits use case	Writing production code & wanting tests for it [participant reflection on Q16]	Improving test suite and resolving technical debt [participant reflection on Q16, Chapter 2]
Understand coverage contribution and test execution	In targeted method in detail [Q4, design user-guided test amplification]	Across the whole project [Chapter 3]
Expectation of receiving tests	Might disappoint if targeted branch cannot be covered [Technical study]	Only proposes tests / additional coverage it can provide [design open test amplification Chapter 2]
Runtime efficiency	More effective at providing tests for method of interest [Technical study]	Can provide larger coverage variety of tests for whole class [Chapter 2]

The two amplification approaches **fit two complimentary use cases** for software developers. From the participants reflecting on which approach is more helpful to generate tests (Q16), we learned that the user-guided version is better suited when they write tests in conjunction with the production code, also called test-guided development [8, 179]. When their focus is to improve the test suite itself, e.g., to address technical test debt [180–183], open test amplification would be the better choice. This is because it connects an amplified test clearer to the original test from the test suite by pointing out the applied input modifications.

Open test amplification also informs the developer about the **coverage impact of an amplified test** across the whole project [Chapter 2]. With the high prevalence of integration tests in JUnit test suites [43, 44], tests amplified from them can improve test coverage in several locations throughout a software project [Chapter 3]. Because this scattered coverage information can be confusing [Chapter 2] and partially irrelevant to developers [Chapter 3], user-guided test amplification focuses only on the impact in the targeted method. In return, it can use the available room to convey the stronger metric of branch coverage in a simple and easy to understand visualization (Q14).

A previous study on the interaction of software developers with test amplification showed the importance of **managing the users' expectations** and making sure they align with what the tool can provide [Chapter 2]. Open test amplification only proposes tests for locations it can actually cover, so it can easily fulfill the user's expectations for receiving tests. In our proposal of user-guided test amplification the developers can select any branch as a target, but as we saw in the technical study, more than half of the branches in our study projects could not be covered. This might disappoint the user and not meet their expectations. When the participants of our study encountered this, they however were positive about the fact that the tool clearly reported that it could not generate a test (participant reflection on Q12). To address the low success rate of guided test amplification, we would need to initialize the objects and parameters correctly to hit the targeted branch (manual inspection technical study). Advanced techniques like concolic execution [184–186], or search-based optimization [187] could address this. However, these can be expensive to compute.

When studying the **effectiveness** of test amplification in our technical study, we saw that guided test amplification produces a higher ratio of tests that successfully cover the targeted branch. This highly fits the use case of testing the developer's current focal method. In contrast, the more explorative search in the whole method space of a class under test that open test amplification performs is more effective when the goal is to improve the coverage across the whole class. Someone who uses guided test amplification for this would need to invoke it over and over again for each method in the class.

5.4.1 Implications for Practitioners

Our evaluation of user-guided and open test amplification uncovered a set of trade-offs a software developer or their manager should consider when choosing which approach to apply. The main, reoccurring consideration is *why* someone wants to generate tests: (1) to improve the test suite itself (choose open test amplification), or (2) to get support for writing tests while working on a specific part of the production code (choose user-guided test amplification). Beyond this, our study also shows anecdotal evidence that when a code

base contains many complex classes with private constructors, test amplification with our state-of-the-art tool will likely not be able to cover many branches.

5.4.2 Implications for Researchers

For researchers in the area of test amplification and generation, as well as developer-centric support tools, the insights from our study point to several new research directions.

Improving the effectiveness of guided test amplification asks for more advanced techniques to initialize objects to cover the targeted branch. Can we apply computationally expensive techniques while still providing an interactive user experience?

Could we actively ask the developer to help us with the initialization of objects that are hard to create? Here the question is whether they would know enough to provide a valuable initialization and whether the automation would still be worth it to use for the developer if they would have to contribute such substantial effort to the test generation.

Many decisions in the design of either test amplification approach are motivated by the required for a response time that feels interactive to the user. Would it be feasible to pre-generate tests in the background and then selectively present relevant ones to the developer when they request tests? A complication here is that current developer-test generation approaches like test amplification or search-based generation with EvoSuite [17], require the code under test to be available. However, we observed repeatedly in our user study that developers are looking for tests covering the code they just wrote a short while ago.

Why did the participants of our study prefer the control-flow graph visualization of the branch coverage over the bytecode instruction visualization of the line coverage? Based on our observations, we conjecture that the following aspect could influence this: (1) using a coverage metric that is embedded in the developer's mental structure of the code, (2) limiting the scope of the displayed code coverage to just the one method the developer is concerned about, and (3) presenting the existing coverage in conjunction with the additionally provided coverage, letting the developer grasp the differential impact a new amplified test makes.

5.5 Related Work

In this section, we discuss related work from the areas of directed and interactive test generation.

5.5.1 Directed Test Generation

Search-Based Software Testing (SBST) uses search algorithms to automatically find tests that a variety test objectives captured in a fitness function [188]. SBST has been used to automate test generation for various test goals, such as maximizing structural coverage [10, 116, 189–191] and crash reproduction [23, 187, 192].

Test suite augmentation techniques are used to generate tests that target code changes that the existing test suite does not cover [121]. Xu et al. proposed several approaches for test augmentation using concolic testing [122], genetic algorithms [193], and a combined, hybrid approach [194, 195]. In their concolic approach, they find the source node of a changed branch and select existing tests that reach this source node. Then they explore different directions of path conditions to find new tests for the changed branch. Their

genetic algorithm uses a fitness function that prefers the distance of a test's execution to the changed branch. In contrast to their approach, our test amplification focuses on all uncovered branches of a software, not just the recently changed ones. Further, our approach is simpler, as we only select a few initial tests and only amplify them with one evolutionary iteration.

Several researchers focused on generating targeted tests to support debugging. Ma et al. propose directed symbolic execution, using the distance to the target line as information to guide the symbolic execution [196]. Dinges et al. [197] combine symbolic execution, to find a suitable entry point to reach a target statement, with concolic execution and heuristics, to try to satisfy constraints too difficult for the symbolic execution. Our approach makes use of the existing tests as a basis for the amplification, and we do not use symbolic execution to reduce our computational costs.

5.5.2 Interactive Test Generation

Several techniques are discussed to incorporate information provided by humans into the test generation process. Marculescu et al. proposed Interactive Search-Based Software Testing (ISBST) to involve domain specialists in test generation [77]. Their feedback adapts the fitness function during the search process by changing the relative importance of system quality attributes. The primary difference between their work and ours is that they involve domain specialists in the test generation, while we target software developers. They pointed out the importance of perfecting how automated test systems communicate with users and ensuring that results are understandable to the users when transferring ISBST to industry [78]. We address this in the design of our interface, visualizing information about the test amplification results to help the user's comprehension.

Murphy et al. propose to apply grammatical evolution into SBST and incorporate human expertise into the search [198]. They proposed that users can define the search space they want their tests to be created from by specifying a grammar. Ramírez et al. observed two key issues hindering the acceptance of automated tests by analyzing various studies that evaluated the effectiveness and acceptance of test generation tools [174]: the opacity of the generation process and the lack of cooperation with the tester. To address this, they incorporate the tester's subjective assessment of readability to compare tests with the same fitness in a search-based test generation process. Our work also addresses the concerns Ramírez et al. raised. We cooperate with testers and make the process transparent by letting testers express their branch coverage goal and guide the test generation. We also improve the understandability of tests by connecting the amplified tests with testers' coverage goals.

5.6 Conclusion and Future Work

The aim of user-guided test amplification was to ease the effort for software developers when understanding amplified tests, by letting them point the test generation to a specific target branch and then visualizing the resulting coverage leveraging a control flow graph of the method under test. Through our technical case study, we show that even simple modifications to the amplification process make guided test amplification more effective at generating tests for a targeted branch. Our user study shows that developers prefer the

interaction with user-guided test amplification, but that the choice for either technique is dependent on the current use case of the developer. From our studies and the design of both approaches, we identify and discuss four trade-offs that influence the choice between open and user-guided test amplification: (1) the current task and goal of the developer, (2) where the amplified test should provide coverage, (3) the ability to fulfill the user's expectation to receive a generated test, and (4) the available time for the test amplification.

Beyond the research implications we mentioned earlier, our work can be the basis for several future research directions:

We observed the developer's wishes to generate tests while they are working on a particular piece of code. While user-guided test amplification is a step in this direction, the next step would be to detect when a developer has finished a change, and automatically generate and propose a test for the code change to the developer.

The feedback on the coverage visualization showed that it helps developer to understand test coverage better. On the other hand, the expectations of the user guiding the amplification now requires more advanced test generation approaches that are already available in other tools. The next step, would be to disconnect the test generation tool from the interaction layer that proposes the tests to developers. This allows for more flexibility in choosing the test generation tool that is right for the job while still benefitting from the continued advancement in test communication.

6

Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps

Can fuzzers generate partial tests that developers find useful enough to complete into functional tests (e.g., by adding assertions)? To address this question, we develop a prototype within the Mozilla ecosystem and open 13 *bug reports proposing partial generated tests for currently uncovered code. We found that the majority of the reactions focus on whether the targeted coverage gap is actually worth testing. To investigate further which coverage gaps developers find relevant to close, we design an automated filter to exclude irrelevant coverage gaps before generating tests. From conversations with* 13 *developers about whether the remaining coverage gaps are worth closing when a partially generated test is available, we learn that the filtering indeed removes clearly non-test-worthy gaps. The developers propose a variety of additional strategies to address the coverage gaps and how to make fuzz tests and reports more useful for developers.*

This chapter is based on *C. Brandt, M. Castelluccio, C. Holler, J. Kratzer, A. Zaidman and A. Bacchelli. Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps, IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2024* [56].

While the importance of automated tests is widely accepted [58, 199], creating them is a tedious task for developers [6–8]. Automatic test generation aims to alleviate the developer's effort when writing tests. State-of-the-art tools can reach high structural coverage [10, 13, 47, 57], [Chapter 2], but face obstacles like understandability of the tests [35, 159], [Chapters 3 and 4] and integration of the tools into the companies tooling [45, 200]. In contrast, automated testing tools from the security community, namely fuzzers, are successfully applied in practice[1] [201, 202]. Fuzzers explore possible inputs to a program to find crashes and potential security vulnerabilities [5, 128]. At Mozilla, fuzzers find around 25 % of all critical or high rated security vulnerabilities, year after year (see Figure 6.1). In this experience report, we document our exploration on whether fuzzers can generate partial tests that developers find useful to complete into functional tests.

One of the sources of effort when writing tests is to create the fixture—the setup and operations needed to reach the targeted code to be tested [203]. This is where fuzzers can help: When they find a crash, they return the fixture and inputs triggering the crash. To turn the fixture into a complete functional test, a developer would then add an assertion that checks the behavior of the code under test. Let us illustrate this with an example:

> To improve their test suite, Ezra's software company introduced a tool that proposes partial fuzzing-generated tests to developers. From the tool, Ezra receives an issue report including a fuzzing-generated fixture that reaches a line of code that is not yet covered by their test suite. She inspects the targeted code and the provided fixture to judge whether the fixture is useful enough for her to complete it into a functional test. To complete it, Ezra adds an assertion that checks the behavior of the targeted code, surrounds it with their test framework's template, and then includes it in their test suite.

To explore the potential of such a tool, we conduct a study with two main phases. First, we build a prototype based on Mozilla's fuzzing infrastructure. Using this, we submit issue reports with partial tests that draw from the output of fuzzers. We analyze the responses to the reports to answer our first research question:

RQ1: What are developers' reactions when proposing fuzzing-based tests to be completed into functional tests?

Our goal is to determine whether developers see enough value in these tests to develop them into functional tests, such as by adding assertions. To enhance the significance of these partial tests, we tailor them to target code sections not covered by current tests. Based on developer feedback, we observe that they first assess the significance or "relevance" of the specific code under test before evaluating the value of the partial tests themselves.

Drawing from these insights, in the study's second phase, we design a filter to pinpoint coverage gaps more relevant to the developers. We then engage with developers to gauge their interest in addressing these gaps, addressing our second research question:

[1] https://hacks.mozilla.org/2021/02/browser-fuzzing-at-mozilla/

RQ2: What are developers' opinions about closing the coverage gaps remaining after our filter?

Our study reveals developers' criteria for determining the significance of closing a coverage gap and their preferred methods, including completing our partial fuzzing-based tests.

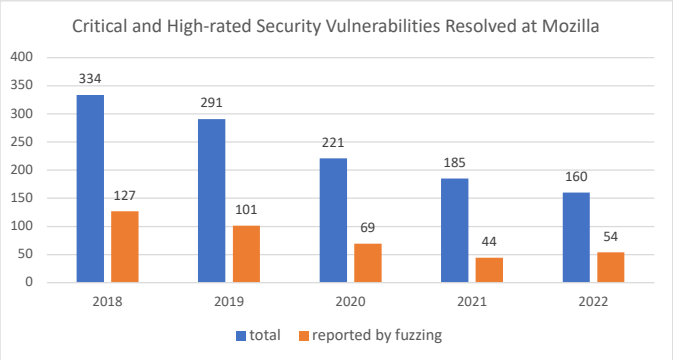


Figure 6.1: Number of resolved critical and high-rated security vulnerabilities over the years at Mozilla. See Section 6.8 for how we obtained these numbers.

6.1 Fuzzing To Inspire Functional Tests

In this section, we explain our approach to generate partial, fuzzing-based tests and to create reports for developers to complete them to functional tests. We start by describing our general design, and then explain how we concretely implement it for Mozilla.

6.1.1 Inspiration Through Fuzzing-based Tests

The test generation consists of five steps, which we illustrate in Figure 6.2.

First, we obtain the line coverage of the current regression test suite, e.g., from the continuous integration artifacts ①. We then instrument all code blocks which are not executed by the test suite, i.e., they are not yet covered by the tests ②. We instrument these blocks with an assertion that will trigger the fuzzer: When a fuzz test executes one of our inserted assertions, the fuzzer will register a crash and save the fuzz test. In the third step ③, we use a generative fuzzer to collect tests that execute the instrumented, not-yet-tested code blocks. During the fuzzing, we also collect auxiliary information such as which instrumented code block is executed and the stack trace when the fuzzer hit the assertion. Next ④, we minimize the fuzz tests to only those lines necessary to trigger the instrumented assertion. When there are multiple minimal tests, we keep them all to provide alternative tests to the developer.

After obtaining the minimal fuzz tests, we use them and the auxiliary information to create a bug report for developers ⑤. We explain that we have a partial test and link to the untested code block it executes. We provide the test with the smallest character count,

and attach the stack trace to help the developer understand how the fuzz test executes the code block. We also attach the other minimized tests as alternative inspirations for the developer. The bug report asks the developer to complete the test by adding a functional check, in xUnit terms: an assertion. If they think it is worth to do so, the developer should add the test to the regression test suite. For this, they also need to write the boilerplate code necessary to integrate the test into the test suite. Figure 6.4 shows an example test from the Firefox source code, highlighting the functional assertions and the boilerplate code that embeds the test into the test suite. The fuzzing-based tests don't contain the boilerplate code yet, because the fuzzer uses a different harness to execute the tests, and there are multiple possible test suites with their own frameworks that the developer might choose to add the test to.

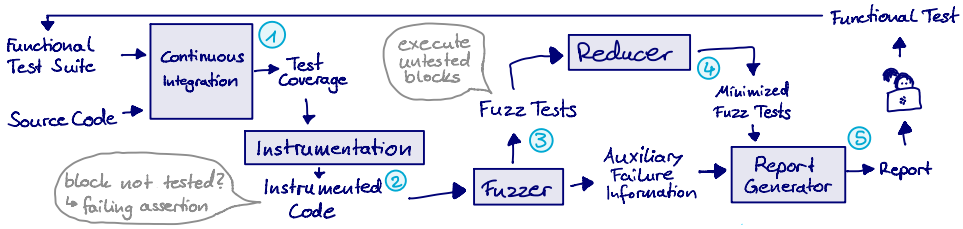


Figure 6.2: Overview of our fuzzing inspiration approach.

6

```
<script>
window.requestIdleCallback(window.close, {timeout: 10000})
</script>
<style>
html:last-of-type, #htmlvar00001 {
  text-align-last: start; }
.class0, aside:nth-last-child(2) {
  column-width: 1em;
</style>
<table>
<colgroup width="3" span="20">+GEE&gt;uo/c(wt6,N:1=</colgroup><caption class="class0">
```

Figure 6.3: A partial fuzzing-based test produced in our study. To complete it, a functional assertion and test framework boilerplate code needs to be added.

6.1.2 Instantiation in the Mozilla Ecosystem

We implemented our approach for the development environment of the Mozilla Firefox browser. Several regression test suites are based on `.html` files that are executed in a sandboxed browser environment [204]. Figure 6.4 shows an example test that checks that the browser automatically scrolls to the right anchor on a page. The functional check happens in the `is(...)` call, where two values are compared and if they are not the same, the test fails with the provided error message. To generate partial tests matching these browser tests, we use the Domato fuzzer.[2] Domato is a state-of-the-art generative DOM fuzzer, that uses grammars to generate random HTML, JavaScript, and CSS code in one

[2]https://github.com/googleprojectzero/domato

`.html` file [205, 206]. We use Mozilla's `grizzly`[3] harness to run Domato's generated tests in Firefox and record crashes and auxiliary information. We employ delta-debugging [207] implemented in the tool `lithium`[4] to reduce the fuzz tests to the minimum lines needed to trigger the crash—to execute the instrumented, not-yet-tested code block. For the exact configurations, please refer to our replication package.[5] Figure 6.3 shows one example of a reduced fuzz test generated during our case study. The reports we generate are for Mozilla's issue tracker Bugzilla and use their code search engine Searchfox to link to the targeted code block. As an example, Figure 6.5 shows an one of the reports we generated during our study.

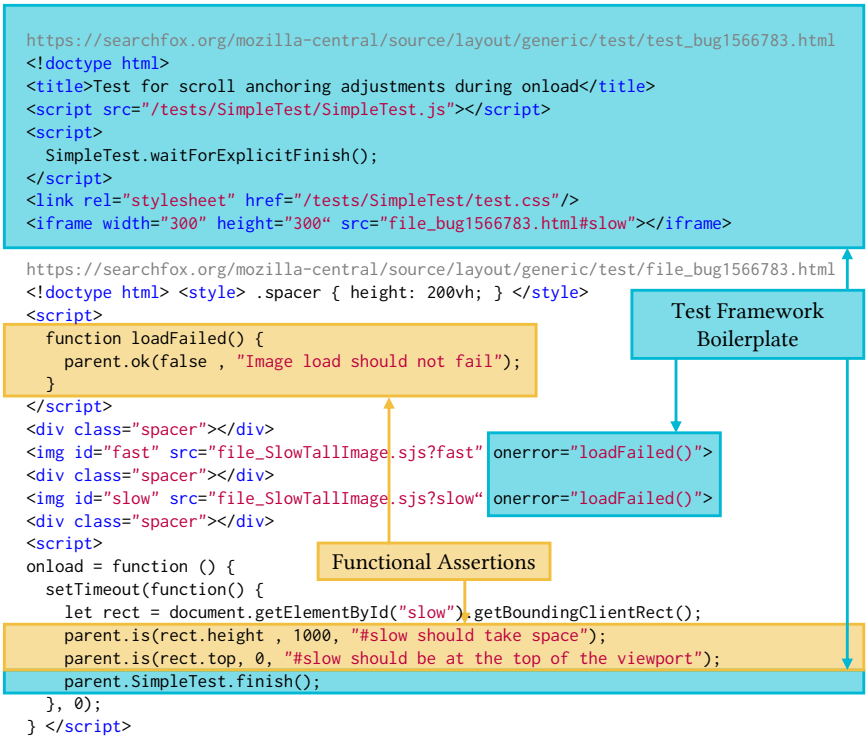
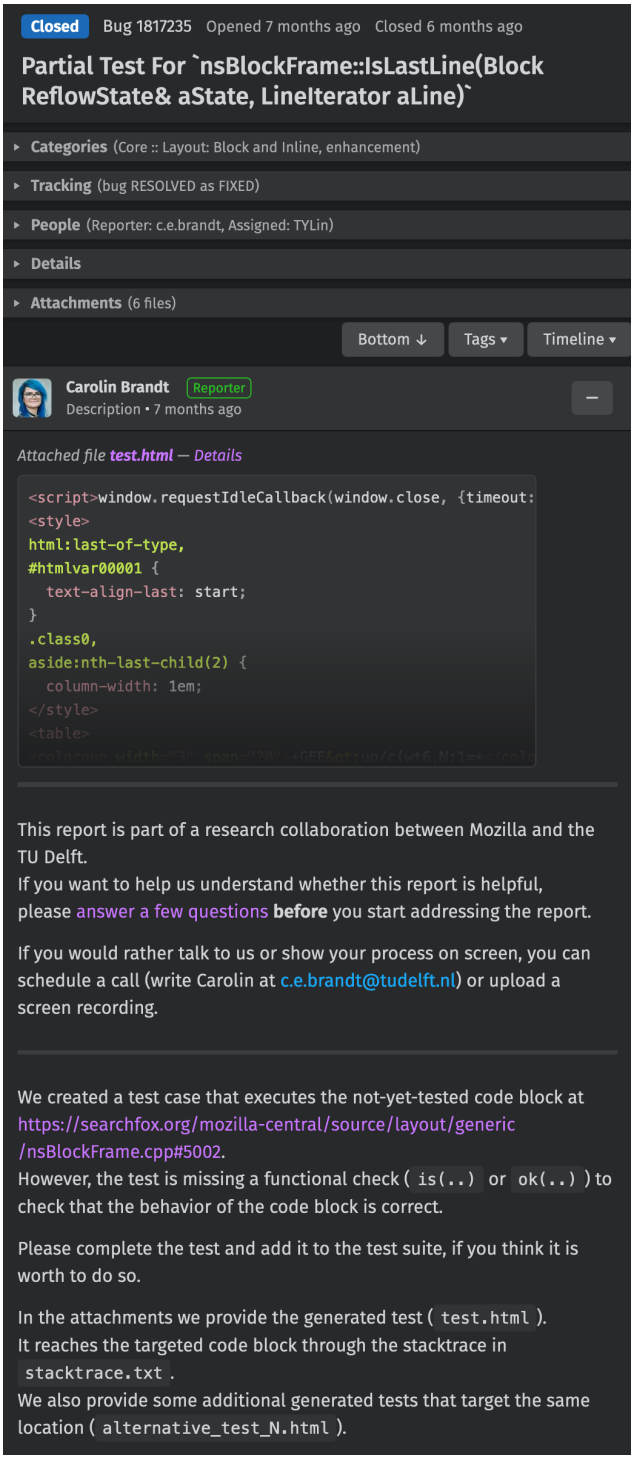


Figure 6.4: A test from the Firefox regression test suite.

[3]https://github.com/MozillaSecurity/grizzly
[4]https://github.com/MozillaSecurity/lithium
[5]You can browse our replication package at: `https://anonymous.4open.science/r/moz-fuzz-inspiration-replication/readme.md` or download it at `https://zenodo.org/doi/10.5281/zenodo.10470823`.



Closed Bug 1817235 Opened 7 months ago Closed 6 months ago

Partial Test For `nsBlockFrame::IsLastLine(Block ReflowState& aState, LineIterator aLine)`

- **Categories** (Core :: Layout: Block and Inline, enhancement)
- **Tracking** (bug RESOLVED as FIXED)
- **People** (Reporter: c.e.brandt, Assigned: TYLin)
- **Details**
- **Attachments** (6 files)

Bottom ↓ Tags ▾ Timeline ▾

Carolin Brandt Reporter
Description • 7 months ago

Attached file ***test.html*** *— Details*

```
<script>window.requestIdleCallback(window.close, {timeout:
<style>
html:last-of-type,
#htmlvar00001 {
  text-align-last: start;
}
.class0,
aside:nth-last-child(2) {
  column-width: 1em;
</style>
<table>
```

This report is part of a research collaboration between Mozilla and the TU Delft.
If you want to help us understand whether this report is helpful, please answer a few questions **before** you start addressing the report.

If you would rather talk to us or show your process on screen, you can schedule a call (write Carolin at c.e.brandt@tudelft.nl) or upload a screen recording.

We created a test case that executes the not-yet-tested code block at https://searchfox.org/mozilla-central/source/layout/generic/nsBlockFrame.cpp#5002.
However, the test is missing a functional check (`is(..)` or `ok(..)`) to check that the behavior of the code block is correct.

Please complete the test and add it to the test suite, if you think it is worth to do so.

In the attachments we provide the generated test (`test.html`).
It reaches the targeted code block through the stacktrace in `stacktrace.txt` .
We also provide some additional generated tests that target the same location (`alternative_test_N.html`).

Figure 6.5: A Bugzilla report submitted during our study.

6.2 Proposing Inspirational Fuzzing-Based Tests To Developers

To investigate the feasibility of our approach for proposing partial, fuzzing-based tests for completion to developers, we conduct a prototype study [208] at Mozilla. Our goal is to explore whether our approach can provide tests that are helpful to software developers, and what aspects require further attention to create tests and reports that the developers find useful. In the study, we generate tests for uncovered code in the Firefox code base and submit 13 Bugzilla reports proposing them to developers. We analyze the ensuing discussions on the reports to identify why the developers choose to act on the report or not, and how they resolve them. We conducted a risk analysis and sought approval from our local ethics review boards with respect to data protection.

6.2.1 Study Design and Execution

For our study, we choose two folders in the Firefox code base to instrument and generate tests for. The folder `/dom` contains the code pertaining to the implementation of the Document Object Model[6] (DOM) and its APIs. The folder `/layout` contains the layout engine, responsible for laying out the elements of the page in the correct positions.[7] In initial trials, we saw that when inspecting the fuzzing-based tests generated for these folders, we could draw clear connections between the objects and attributes in the test generated by the DOM fuzzer and the code targeted by the tests. For this pragmatic reason, we opted to focus on these two folders.

After instrumenting the code,[8] we let our fuzzer run on a desktop machine for 30 minutes. This yielded us 133 fuzz tests and corresponding reports. Of these, 36 were duplicates targeting the same coverage gap. To not overwhelm the particular contributors or groups, we decided to only open one Bugzilla report per Firefox component. Components are categories of functionality that Mozilla uses to manage responsible reviewers and triagers within Bugzilla. Each code file belongs to one component, and we identified the corresponding component by looking at the file which contains the coverage gap targeted by a generated test. We submitted 13 Bugzilla reports, one for each of the components present in our set of generated tests. In Figure 6.5 you can see an example of such a report. We provide the shortest test generated by the fuzzer for the targeted coverage gap and the stacktrace showing how the test reaches the coverage gap. In addition, we included alternative tests that the fuzzer produced for the same coverage gap, to provide more options in case the shortest test was not useful.

Over the following week, we responded to any comments and questions by the developers. Because we categorized our reports as enhancements, many were initially not picked up through the triage processes focusing on reports labeled as defects. After identifying relevant developers based on the authors or reviewers of the patches that created the code under test, we pinged them personally and then received reactions to four more of the reports (1817159, 1817173, 1817235, 1817219).

[6] `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`
[7] `https://wiki.mozilla.org/Platform/Layout#About`
[8] Revision: 63a3d733b2331033f48d10995ce09abf50def953 in `mozilla-unified`

Bug ID	Component	Resolution
1816640	SVG	Open
1816862	DOM: Serializers	No Reaction
1817150	CSS Parsing and Computation	No Reaction
1817154	DOM: Core & HTML	No Reaction
1817158	Layout	No Reaction
1817159	XSLT	Open
1817215	MathML	Open
1817221	Audio/Video	No Reaction
1817238	Layout: Tables	No Reaction
1817173	Web Painting	Open
1817176	DOM: Networking	Resolved
1817235	Layout: Block and Inline	Resolved
1817219	Layout: Images, Video, and HTML Frames	Resolved

Table 6.1: List of all Bugzilla reports we submitted, including the targeted component and the resolution state of the report. The reports can be accessed via `https://bugzilla.mozilla.org/show_bug.cgi?id=<id>` and are hyperlinked in the ID column.

6.2.2 Developer Reactions

In this section, we give an overview of the reactions to the reports we submitted. Table 6.1 lists each report, the component it is related to, and a link to the full discussion. Out of the 13 reports, six received no reaction, three were resolved, and four received comments but remain open.

From the reports that received reactions, most prominently the developers focused on **whether the targeted code is worth testing**. The developer reacting to report 1816640 stated that: "All [method under test] does is forward to setAttribute('type', value). I'm not sure that there's much value in testing it as it's only one line of code." Two developers reacted to report 1817159. One pointed out that the code for this feature is "rather derelict", but then pointed to a recent zero-day bug related to this code, stating that "maybe it would be good to invest a bit of time ensuring that we have good testing". A second developer later explained that, together with a colleague, they determined that the targeted code is suitable to write tests. However, they also point out that these tests "wouldn't teach or touch deeper xslt logic", which has the team's priority at that moment. We interpreted "teach" in the vein of tests serving as documentation on how to use the code under test [40, 58, 108]. Even though tests for the targeted code would "improve exception catching in tests without doubts", they decide to leave this in their backlog. The developer reacting to report 1817219 explained that each of the tests are expected to trigger the early return in the targeted coverage gap. They stated that it would not be worth to test that code, also because they recently changed the surrounding behavior.

When the reports lead to action from the developers beyond comments, we observed a variety of **ways to address the reports**. Report 1817176 received a quick reaction with a patch submitted to code review. The patch did contain a test for the targeted coverage gap, inspired by the test we submitted, but in a different format than our proposed one: instead of a `.html` file, it was an addition to a `.json` file that configures parameterized tests.

The first developer reacting to report 1817235 stated that it "looks potentially interesting," explaining the behavior triggered by the test. They propose to clean up the provided test cases, making sure that the behavior of the code actually makes sense. A different developer picked up the task and submitted a patch for the targeted line of code. However, when the developer worked on the report, our tests did not trigger a `printf` statement they added in the while loop they targeted to test. The developer speculated that this could have been caused by changes to the code under test between the moment of fuzzing, opening the bug report and addressing the bug report. So they crafted a test case themselves, inspired by hints from our provided tests on which attributes are involved in triggering the code under test. Report 1817219 was resolved by submitting and merging a patch that removed the whole method targeted by our fuzz test. One developer linked our report 1817215 to another active report about updating tests for this component after changes in the cross-browser web-platform-tests.

We received a very detailed reaction to report 1817173, analyzing the behavior triggered by the generated tests. The targeted coverage gap was a fallback path that is no longer used because a newer component has taken over its responsibility on the operating systems used for the CI coverage data. The developer pointed to the line of test code that the newer component could not handle and which triggered the fallback path. They initially propose two solutions to resolve the bug: (1) adding a test suite variant that runs all existing tests while enforcing the fallback path, or (2) duplicating some related tests and modifying them to use the fallback path. In the ensuing discussion, we uncovered that the former option already exists, but this **test suite covering the targeted code is not executed in the current CI** coverage calculation.

RQ1: Developers' reactions when proposing fuzzing-based tests to be completed into functional tests

The developers reflected on whether the code targeted by the tests is worth covering, mentioning a too small coverage gap or early returns as reasons to not act upon the reports. Another reason was that the code was covered by tests not executed on the CI. Other reports were addressed in a variety of ways: submitting a syntactically different test for the same scenario, using our test as a starting point to write their own test, or removing the targeted code as it was no longer used.

6.3 Selecting Relevant Coverage Gaps

In the developer's reactions to our Bugzilla reports, we observed that several of the coverage gaps we targeted with the fuzzing-based tests were considered less relevant to test by the developers (Bug 1816640, Bug 1817159, Bug 1817219). This led to the developers not further acting upon our generated tests. To provide more relevant reports and partial tests, we decided to take a deeper look at which coverage gaps we should target with our tests.

For this, we designed an automatic filtering step that excludes less interesting coverage gaps (between ① and ② in Figure 6.2) before instrumenting the code in preparation for the fuzzer. The pseudo-code for the automatic filter is shown in Figure 6.6. The filter excludes both:

- coverage gaps that are only a single line of code long, as the comments on the bug reports deemed such gaps as too small to close (Bug 1816640), and
- coverage on conditions and branches that represent an early return out of a function (Bug 1817159, Bug 1817219).

What we call an early return is when a function returns a default value after checking for an error or warning. We detect such early returns by keywords, such as `NS_ERROR` or `Throw`, combined with a `return` in the coverage gap. We manually inspected further coverage gaps that were hit early, i.e., in two 2 minute runs, by our fuzzing approach, and extended the keyword list to detect coverage gaps that were deemed as early returns by our Mozilla collaborators. When running our filter over the coverage gaps in the folders we instrumented before (`/dom` and `/layout`), we exclude 15 054 (single line) and 8085 (early return) coverage gaps, leading to a remainder of 8644 coverage gaps to instrument.

```
def should_we_instrument_this_line():
  if (len(coverage_gap.lines) == 1)
  or (len(coverage_gap.lines) == 2
    and second_line_contains_only_closing_brace)
  or is_early_return():
    return False # don't instrument

def is_early_return():
  is_error_or_warning, is_return = False
  if "NS_WARN_IF" in condition_before_coverage_gap:
    is_error_or_warning = True
  for line in coverage_gap.lines:
    if ("NS_WARNING" in line
        or "NS_ERROR" in line
        or "Throw" in line
        or "WEBM_DEBUG" in line
        or "Error" in line
        or ("promise" in line and "reject" in line))):
      is_error_or_warning = True
    if "return" in line:
      is_return = True
  return is_error_or_warning and is_return
```

Figure 6.6: Our filter for interesting coverage gaps.

6.4 Do Developers Think These Coverage Gaps Should Be Tested?

To evaluate whether our filtering for coverage gaps indeed yields more interesting test targets, we again reach out to the developers for feedback. We retrieve a new, more recent revision and CI coverage run of the Firefox code base[9] and apply our filter to select relevant coverage gaps. To extend our reach of potential developers to talk to, we extended the folders of the source code we implemented to include `/accessible`, `/editor`, and `/gfx` in addition to `/dom` and `/layout`. Applying the filter left us with 19 050 coverage gaps to instrument, after excluding 30 672 (single line) and 13 626 (early return) coverage gaps. We instrumented the remaining coverage gaps and again generate fuzz tests for 30 minutes on

[9]Revision: 0bcf2642f5a6e7175812623451eda2ab6cb35a0d in `mozilla-unified`

a desktop computer. This yielded us a set of 44 coverage gaps that pass our relevance filter and could be hit by the fuzzer within the short time budget of 30 minutes. We manually validated if each coverage gap is still present and whether it should have been filtered: We exclude five false positives (three early return statements, two single statement coverages formatted to span two lines) and one coverage gap inside code that is only executed during fuzzing runs.

We identified developers that can likely judge the test-worthiness of the coverage gaps by looking at the authors and reviewers of the patch that introduced the targeted line of code or recent patches the surrounding lines. Patches that did not show experience with the code at hand, e.g., large scale refactorings, and contributors no longer with Mozilla, were excluded by us. The second author, who is the manager of the CI and Quality Tools team, which owns the coverage infrastructure and other development tools, contacted 13 developers. We briefly explained our project and that we are now trying to identify areas of code that are interesting to cover with a test. We then gave them one or more code locations and asked whether they can explain why or why not they are valuable to test. Additionally, we sought their consent for using their comments in the paper publishing this study.

Two authors analyzed the chat conversations by independently applying open and axial coding. Then, they compared and merged the emerging themes from their independent analyses. In the following, we will describe our observations along three major categories: First, we take a look at the developers' rationales for why a coverage gap is worth testing or not, motivating the need for a more refined way of looking at code coverage and the need to close coverage gaps. Then, we present varied proposals from the developers on how to address the coverage gaps in other ways than completing the fuzz test to a functional one. Finally, we consider our proposed approach of submitting bug reports with fuzz tests to be completed to functional tests and discuss feedback from the developers on how to modify the tests, report, and workflow to better fit their needs. Table 6.2 lists the coverage gaps we discussed with each of the 13 developers, identified in the following by D1–D13. For example, with D4 one of the four coverage gaps we discussed were lines 90–105 of `dom/svg/SVGMotionSMILAnimationFunction.cpp`.

6.4.1 Test Relevance of the Filtered Coverage Gaps

Our deeper look at filtering for interesting coverage gaps was motivated by the feedback on our initial Bugzilla reports pointing out coverage gaps too small to be worth closing (1816640), and fallback options that return early from a method in case of an error (1817173). Looking at the conversations, the filtering seems to be effective as we did not receive answers along the lines of "this is too simple code to be worthy a test." D1 reflects on a very particular reason why the code is not tested at the moment. They explain that "our tests only test the successful part", which is a potential sign of confirmation bias [209]. Furthermore, D1 states that the specific failure handled by the targeted code is caused by operating systems and hardware not available on the current CI servers.

In the conversations, the developers gave **reasons why some coverage gaps should be closed**: to catch regressions (D11), increase the confidence during rewrites and larger-scale refactorings (D3, D8), documenting edge case bugs (D3), testing important edge cases (D10, Figure 6.7), or ensuring that the behavior matches an external specification (D2, D4).

D1	dom/media/platforms/PDMFactory.cpp#678-680 dom/media/platforms/PDMFactory.cpp#725-727
D2	dom/svg/SVGFEImageElement.cpp#191-196
D3	dom/events/ContentEventHandler.cpp#1752-1755
D4	dom/svg/SVGMotionSMILAnimationFunction.cpp#90-105 layout/painting/nsCSSRendering.cpp#4112-4113 dom/svg/DOMSVGLength.cpp#297-298 layout/base/PresShell.cpp#9665-9667
D5	layout/painting/nsCSSRendering.cpp#2418-2425 layout/generic/nsBlockFrame.cpp#1133-1142 dom/html/HTMLSharedElement.cpp#96-102 layout/style/GeckoBindings.cpp#1397-1401 dom/svg/SVGStyleElement.cpp#164-167 layout/base/PresShell.cpp#9665-9667
D6	layout/painting/nsCSSRendering.cpp#4112-4113 layout/svg/SVGTextFrame.cpp#3922-3926 gfx/thebes/gfxFont.cpp#1530-1533 layout/svg/SVGTextFrame.cpp#2881-2883
D7	editor/libeditor/EditorBase.cpp#2888-2890
D8	dom/svg/SVGFETileElement.cpp#51-55 layout/painting/nsCSSRenderingBorders.cpp#2845-2847 layout/painting/nsCSSRenderingBorders.cpp#2745-2748
D9	dom/xslt/xslt/txMozillaXSLTProcessor.cpp#881-890 dom/xslt/base/FtxDouble.cpp#52-53 dom/base/DirectionalityUtils.cpp#613-619 dom/base/DirectionalityUtils.cpp#1069-1070 dom/base/DirectionalityUtils.cpp#339-341
D10	layout/base/PresShell.cpp#9665-9667
D11	dom/base/DirectionalityUtils.cpp#613-619
D12	layout/tables/nsTableFrame.cpp#3018-3034
D13	dom/svg/DOMSVGLength.cpp#297-298 dom/svg/SVGElement.cpp#704-706 dom/svg/SVGPathData.cpp#476-481 dom/svg/DOMSVGAngle.cpp#105-107 dom/svg/DOMSVGAngle.cpp#28-30

Table 6.2: Overview of the developer chats and the discussed coverage gaps. The coverage gaps can be viewed via: `https://searchfox.org/mozilla-central/rev/8329a650e3b4f866176ae54016702eb35fb8b0d6/<text in 2nd column>` (also hyperlinked in that column). The given line numbers are the first and last uncovered line of the coverage gap.

```
9663  target->DidReflow(mPresContext, nullptr);                                  5 M
9664  if (target->IsInScrollAnchorChain()) {                                     5 M
9665    ScrollAnchorContainer* container = ScrollAnchorContainer::FindFor(target);
9666    PostPendingScrollAnchorAdjustment(container);
9667  }
```

Figure 6.7: A coverage gap that that should be closed to test important edge cases according to D10. The gap is at layout/base/PresShell.cpp#9665-9667.

D11 motivates that the code in Figure 6.8 should be tested because it uses raw pointers, which are error-prone and may lead to null pointer or use-after-free errors.

On the other hand, the developers gave a variety of **reasons for why the code in the coverage gap is not worth the effort of testing**. One reason is that **they think it is unlikely that there is a bug in the code**, because the function did not change in the last 10 years (D3), no bug reports have been opened in that area for a long time (D3, D8), or it is legacy code that should not receive any changes in the future, as it serves as a fallback to a newer implementation (D8). For the coverage gap discussed with D7, "it's hard to find how to run the path" because they are currently rewriting the component to use the functionality less and less, and planning to eventually remove the code all together. Other coverage gaps are described as **unlikely to be reached** because it is a do-nothing fallback for an error in a third party library (D4, D6, talking about the coverage gap shown in Figure 6.9), or the developers expect the case to rarely happen in practice (D4, D8).

Similar to Bugzilla report 1817173, we again encountered cases where, according to the developer, the **code should actually be covered by tests** (D5 about three coverage gaps, D8 about two coverage gaps). For D8, the separate job running the relevant tests is not executed during the CI runs that calculate the coverage.

6.4.2 Different Ways to Address Coverage Gaps

Throughout the conversations, the developers we chatted with brought up ways to address the coverage gaps that differ from completing the partial fuzz tests we could generate. An overarching concern was whether it would be **easier to manually write a test from scratch** (D4, D6). D13 points out that "it's not hard for a developer that knows SVG to come up with tests that hit those lines". D8 directly starts describing a fitting test scenario for one of the coverage gaps we asked about, and D6 explains "I think this would be simplest to write manually, having identified the relevant code path". D9 stresses that "depending

```
612
613  static nsCheapSetOperator TakeEntries(nsPtrHashKey<Element>* aEntry,
614                                        void* aData) {
615    AutoTArray<Element*, 8>* entries =
616        static_cast<AutoTArray<Element*, 8>*>(aData);
617    entries->AppendElement(aEntry->GetKey());
618    return OpRemove;
619  }
620
```

Figure 6.8: A coverage gap that that should be closed according to D11, because the code uses raw pointers. The gap is at dom/base/DirectionalityUtils.cpp#613-619.

```
4104    // Create a text blob with correctly positioned glyphs. This also updates
4105    // textPos.fX with the advance of the glyphs.
4106    sk_sp<const SkTextBlob> textBlob =                                  2 M
4107        CreateTextBlob(textRun, characterGlyphs, skiafont, spacing.Elements(),  3 M
4108                       iter.StringStart(), iter.StringEnd(),            2 M
4109                       (float)appUnitsPerDevPixel, textPos, spacingOffset);  2 M
4110
4111    if (!textBlob) {                                                    2 M
4112      textPos.fX += currentGlyphRunAdvance();
4113      continue;
4114    }
```

Figure 6.9: A coverage gap that is a do-nothing fallback for a third party library and therefore unlikely to be reached according to D4 and D6. The gap is at layout/painting/nsCSSRendering.cpp#4112-4113.

on how good/bad the generated [tests] are it might be easier to just have someone write them in the first place."

One of the coverage gaps discussed with D6 (see Figure 6.9) was described by them as "would only be used in case of some kind of failure within [a third-party] library", and in that case likely another failure appeared earlier, making the code under test very unlikely to be reached. To validate that this is indeed dead code, they propose to add an assertion that triggers a crash in the regular fuzzing runs, and possibly later an "unreachable" assertion to the production code base to alert the team in case the code does become reachable through future changes. D4 recounted that for one of the coverage gaps the team considered adding a crashtest, but that the value of this would be minimal as the code is already hit by the regular fuzzing runs. These examples indicate that the developers consider the **regular fuzzing runs as an alternative to address missing coverage**, increasing the confidence that these code paths do not lead to crashes or that they are unreachable.

Based on our conversation with D5, three follow-up bug reports were filed. One, the developer filed immediately, discussing an inconsistency between different browser implementations that they discovered by looking at the coverage gap we pointed them to. The report was resolved by adding a cross-browser test for the inconsistency and removing the code causing the inconsistency, including the coverage gap. D5 also asked us to file bugs to remove the code from two of the coverage gaps as the code had become obsolete with a previous change. Together with the reaction to the Bugzilla report 1817219 in our first study, we can see that pointing to coverage gaps can also nudge developers to **delete code that became obsolete**.

6.4.3 Needs of Developers and How to Improve Our Approach

In our opening messages to the developers, we pointed to our project of generating tests for the coverage gaps we asked about. Because of this, several of the developers also reflected on the usefulness of such tests and the process of proposing them. D11 was open to try out the generated tests, but stressed that the test should **conform to the common test frameworks** in the project. They also proposed to directly add the test as a patch to the code review platform where the developers can edit the test assertions. D3 stated that it would be more useful to **receive the test at the time of writing the patch** that

introduces the targeted line of code, as compared to receiving the tests weeks later.

Several developers saw some value in providing a generated test alongside pointing to the coverage gap. D6 explained that it is useful to know that a piece of uncovered code could be covered, the **generated test can prove that the code is reachable**. They also describe that the information about the coverage gaps can surface which "combinations of features are going untested at present." With D12 we discuss a coverage gap that they described as a condition "not going through the normal … process." We provided D12 with our generated fuzz test and they opened an issue with it to "use the test case as a start point to investigate if the [covered] branch makes sense or not."[10] The generated tests can also be a starting point and inspiration for a developer familiar with the code to write a complete correctness test case (D4). Three developers (D6, D11, D13) pointed out the **knowledge required to complete the tests** and determine "what the correct behavior … of the test case should be" (D6). This would require familiarity with the domain of the code (D13) or reading specifications to ensure they are followed (D11).

```
2872 std::tuple<EditorDOMPointInText, EditorDOMPointInText>
2873 EditorBase::ComputeInsertedRange(const EditorDOMPointInText& aInsertedPoint,                10 k
2874                                  const nsAString& aInsertedString) const {
2875   MOZ_ASSERT(aInsertedPoint.IsSet());                                                       10 k
2876
2877   // The DOM was potentially modified during the transaction. This is possible
2878   // through mutation event listeners. That is, the node could've been removed
2879   // from the doc or otherwise modified.
2880   if (!MayHaveMutationEventListeners(                                                       10 k
2881         NS_EVENT_BITS_MUTATION_CHARACTERDATAMODIFIED)) {
2882     EditorDOMPointInText endOfInsertion(                                                    20 k
2883         aInsertedPoint.ContainerAs<Text>(),                                                 10 k
2884         aInsertedPoint.Offset() + aInsertedString.Length());                                10 k
2885     return {aInsertedPoint, endOfInsertion};                                               10 k
2886   }                                                                                        10 k
2887   if (aInsertedPoint.ContainerAs<Text>()->IsInComposedDoc()) {                              2
2888     EditorDOMPointInText begin, end;
2889     return AdjustTextInsertionRange(aInsertedPoint, aInsertedString);
2890   }
2891   return {EditorDOMPointInText(), EditorDOMPointInText()};                                   2
2892 }                                                                                           10 k
```

Figure 6.10: The coverage gap we discussed with D7. The gap is at editor/libeditor/EditorBase.cpp#2888-2890.

The **effort required by the developers to complete the tests was seen as problematic** by several of our conversation partners. D11 worries that tests that one needs to spend time to complete will be ignored because "We're already busy enough with intermittently failing tests and what not." For the coverage gap discussed with D7 (see Figure 6.10), they state that it is fine to add a complete generated test, but "that it's not worthwhile to use the developers' time [to write a test] for the edge case."

[10] https://bugzilla.mozilla.org/show_bug.cgi?id=1832450

RQ2: Developers' opinions about closing the coverage gaps remaining after our filter

While our filter successfully excluded coverage gaps clearly not worth covering, several remaining gaps were considered not worth the effort to cover because the developers found it unlikely that there is a bug in the code, the code is unlikely to be reached, or should already be covered by other tests.
Several developers pointed out that it might be easier to manually write a test from scratch than to understand the generated test, and regular fuzzing runs covering code was seen as an alternative to address missing test coverage.
The generated tests can serve as a proof that the targeted code is reachable by a test, but the developers caution about the knowledge and effort required to complete the partial test. To improve our approach, they propose to provide tests that already conform to their common test framework and provide them at the time of submitting the patch with the code under test.

6.5 Discussion

The feedback from the developers indicate that filtering out single-line and early return coverage gaps helps to eliminate "clearly too simple to be worth testing" coverage gaps. Nevertheless, we noted several more reasons that make a coverage gap less relevant for testing. A crucial aspect is the effort required by the developers. Even for coverage gaps described as relevant to test, developers stated that they do not have the time to write a test or complete a generated one, compared to the other tasks they have to complete.

Our initial idea was to relieve parts of the developer's efforts by generating partial tests that reach coverage gaps in their code base. However, we learned to be careful about the additional effort that we put on developer's shoulders when they have to understand a generated test before completing it. In the following, we discuss the implications of our observations for software engineering practitioners, tool builders that want to support them, and researchers in our field.

6.5.1 Implications for Practitioners

In practice, code coverage can be used as a metric by management to judge the quality of testing performed in their teams [143, 210]. The observations in both our studies indicate that **not all "missing" coverage is equally worth testing**. This points to the need for a more refined metric that takes into account the test-worthiness and the required effort to test a coverage gap when measuring the quality of testing. The repeated mention of the effort to understand the generated test and the code under test, as well as the wish to receive the test at the time of writing the code, points to the value of investing in testing at an early stage in development, as the cost of adding the tests later is higher.

6.5.2 Implications for Tool Builders and Developer Supporters

A recurring concern in the second study was the effort to understand and complete the generated tests. When trying to outsource difficult parts of automation tasks like generating

assertions on to human users, we need to make sure that we provide value compared to the user doing the whole task themselves, like writing complete tests from scratch. We saw understanding the generated test is a hurdle to completing it with assertions. With automated generation tools becoming much more popular (e.g., GitHub Copilot and ChatGPT) the work of developers is moving from engineering solutions to evaluating and adapting solutions generated by machines. We conjecture that the next steps need to be to invest in supporting the developers in understanding generated code and tests. One way would be to study and build dedicated tools for this task.

A different option would be to leverage the power of now popular large language models to make the generated fuzz tests more human-readable, or to generate assertions automatically by prompting the model with the generated test and the code under test. To make the generated tests more useful for the developers, we should extend the filter we presented in order to identify code that is worth testing. To reduce the cost of adding a test, we could generate the tests earlier in the development process: at the time the developer is writing the code or a reviewer is reviewing it, reducing the need of context switching.

6.5.3 Implications for Researchers

In both our studies we made initial observations that not all coverage gaps are equally worth testing. This calls for a more detailed study on how to prioritize coverage gaps, what factors influence the test-worthiness of a coverage gap and how to reliably measure these factors. We conjecture that such factors would be very project / company / context dependent. Already in this context we saw that security concerns (point to zero-day bug in 1817159) might weigh stronger than pure code quality improvements (1817159 improving exception testing, but staying in backlog). In addition, we saw hesitancy to touch legacy code that has been running without a link to bugs for long years, as maybe the less risky option to not touch a running system.

Further, in both studies we saw a variety of ways the developers proposed to address our reports or pointers to coverage gaps. Next to writing a test, they also removed code or proposed to add assertions for the regular fuzzing runs to be notified in case the coverage gap becomes reachable. This indicates that **functional testing is not the only way to "cover"/"secure" a line of code**, and metrics we develop to measure the testedness of code should include these other activities.

6.5.4 Threats to Validity

There are several threats to the validity of the observations in both our studies and the conclusions we draw from them. A threat to the *internal validity* is the presence of a social desirability bias, where the developer might have been inclined to answer overly positive about adding tests. To mitigate the impact on our conclusions, we closely report on the developer's statements and the visible actions on the code that followed. While we did receive rationales for why coverage gaps should be closed, we also report on the effort that developers saw and that in many cases led to them not following up and addressing the coverage gap.

Concerning *confirmability*, the threat that the results are shaped by the researcher instead of the respondents, the analysis of the chat conversations with the developers was independently performed by two authors, and we came to a consensus on the observations

and conclusions. These and the summary of the observations from the Bugzilla reports were communicated to and confirmed by the other authors.

With respect to *internal generalizability*, we expect that our observations do generalize to industrial open-source projects and companies with a similar size and positioning towards testing. Looking at *external generalizability*, developers from projects with less familiarity to fuzz testing likely would not point to fuzzing as a way to address the coverage gaps. This and other findings should only carefully be generalized, and we encourage other researchers to replicate our studies in other contexts.

6.6 Related Work

Previous work has studied the introduction of automated test generation tools in industrial contexts. Brunetto et al. [200] report on their experience introducing a tailored automatic GUI test generator in a medium-sized company. A difficulty they faced was the automatic generation of functional oracles, which they mitigated by providing rich reports to support engineers checking the effect of the test on the system. In their lessons learned, they note that automation is welcome in industry, but only useful if the testers can understand and interpret the produced tests. This matches the developer's comments in our study on the additional effort to understand the generated tests compared to writing the tests from scratch. Brunetto et al. report that integrating the output of the test generation into the workflow and tooling of the company was a key factor to enable the adoption of the tool. Further, manually-specified functional oracles would increase the effectiveness of generated test cases. However, a cost-effective way to define these automated oracles (for system-level UI tests in their case) is still an open challenge. In a similar vein, Mesbah et al. [211], who built a tool for automated test case generation for AJAX web applications, note the effort and required knowledge for a developer to specify invariants that can serve as oracles for the correctness of the software behavior. In a survey of 225 software developers, Daka and Fraser [212] find that automated test generation in mainly used with automated oracles, i.e., finding crashes or undeclared exceptions.

Almasi et al. [45] studied the industrial applicability of EvoSuite and Randoop in a financial company. Through generating tests for 25 real faults from the history of the companies' software system and a survey under their developers, they found that in more than half of the cases, more appropriate assertions would have led to the detection of the faults with generated tests. The developers expected the automated generation tools to integrate with their build pipeline and workflow, and were concerned about the readability of the generated tests, input data and assertions. Xie et al. [213] report on their experiences from industrial workshops on teaching testing tools, including the test generator Pex for C# which generates partial tests that developers have to write assertions for. They learned that the developers tended towards staying with the assertion-less automatically generated tests rather than writing assertions for them. Further they pointed to the need to explicitly teach them how they should interact with the generated tests and communicate why the tests were generated with such values.

Zhang et al. [214] present an approach to fuzz test remote procedure call APIs and evaluate it within an industrial context. They report challenges with isolating the test environment (resetting the state of the application, data preparation and mocking of external services) and propose to enable the fuzzer on the CI to promote its adoption

in industry. They also point to the importance of non-flakiness and readability of the generated tests as crucial to be considered when testing industrial APIs. Plöger et al. [125] evaluated the usability of two fuzzers (AFL and libFuzzer) with computer science students. They reported very low usability across all steps of the fuzzing process and gave a variety of recommendations on how to improve, such as UI guidance through the fuzzing process, better error messages, and crash analysis support.

6.7 Conclusion and Future Work

In this chapter, we set out to explore whether partial tests generated by fuzzers and completed by developers can help alleviate the developer's effort of creating tests. For this, we developed a prototype within the Mozilla ecosystem. Through the discussions on 13 Bugzilla reports we created with our prototype, we observed that the code targeted by the tests is a main concern for developers before considering the fuzz tests. More specifically, the developers sometimes indicated that a coverage gap is not worth the effort to be tested. We dove deeper into the test-worthiness of coverage gaps by designing a filter to exclude small and early return coverage gaps. From discussing the remaining gaps with developers, we learned that the filters are effective in excluding clearly irrelevant coverage gaps, but there remain many considerations when deciding whether testing a coverage gap is worth the effort. Remaining gaps were considered not worth the effort to cover because the developers found it unlikely that there is a bug in the code, the code is unlikely to be reached, or should already be covered by other tests. In addition, we saw that there are other ways than functional tests that developers propose to address missing coverage.

Several opportunities for future work follow from our studies. Apart from the aforementioned implications for researchers, the factors of relevance concerning coverage gaps should be studied in other industrial or open source contexts, as we conjecture their priorities to be different between projects and developers. Constructing more reliable coverage calculations, that also include test suites not run on the regular CI runs and other quality assurance or security testing techniques, would provide a more accurate picture of missing coverage and therefore a more reliable basis to guide test generation efforts. Another interesting extension would be to combine the fuzz tests with an automatic approach for assertion generation and study whether confirming a generated assertion reduces the developer's understanding effort far enough to make the approach viable compared to writing tests manually from scratch.

Acknowledgements This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032). A. Bacchelli acknowledges the support of the Swiss National Science Foundation for the SNSF Project 200021_197227. Further support came from the Swiss National Science Foundation (SNSF Grant 200021M_205146).

6.8 Appendix: Fuzzing-Discovered Security Vulnerabilities at Mozilla

The graph in Figure 6.1 is constructed by querying the Bugzilla database for all resolved security vulnerabilities with a critical or high rating on 2023-07-22. For all bugs that were opened in 2018, received a security rating `critical` or `high`, and were resolved, run this query:

```
https://bugzilla.mozilla.org/buglist.cgi?
chfieldfrom=2018-01-01&
chfieldto=2018-12-31&
chfield=%5BBug%20creation%5D&
keywords=sec-high%2C%20sec-critical%2C%20&
classification=Client%20Software&classification=Developer%20Infrastructure&
classification=Components&classification=Server%20Software&classification=Other&
query_format=advanced&
keywords_type=anywords&
resolution=FIXED&resolution=WONTFIX&resolution=INACTIVE&resolution=DUPLICATE&resolution=WORKSFORME&
resolution=INCOMPLETE&resolution=SUPPORT&resolution=EXPIRED&
```

To only see bugs from this search that were reported by the fuzzing team, append to the above query:

```
emailreporter1=1&
emailassigned_to1=1&
emailtype1=exact&
emailcc1=1&
email1=fuzzing%40mozilla.com&
```

To obtain the values for the following years, we changed the year numbers in `chfieldfrom` and `chfieldto`. As this is querying the public Bugzilla database, bugs that are (still) hidden from the public are not included in the results. Focusing on bugs that have been resolved might bias the results towards fuzzing because fuzzing reports are reproducible, which can make them quicker to fix than other bugs that are harder to diagnose or fix.

7

Conclusion

In this thesis, we set out to advance the state of automatic generation of software tests by tapping into the expertise of software developers. To enable this, we worked on generating amplified tests that are useful and helpful for developers. The overarching research question guiding this thesis was:

How do we design an effective collaboration between software developers and automatic test amplification tools?

We explored an initial design of a test exploration tool facilitating this collaboration (Chapter 2), and investigated different facets of this interaction in depth, like communicating behavior and impact of a test (Chapter 3), what changes developers can expect to make to amplified tests when incorporating them into their test suite (Chapter 4), how guidance by the developer towards which code to cover impacts the test amplification process and interaction (Chapter 5), and characterizing the coverage gaps that developers find relevant to close (Chapter 6). In this chapter, we revisit our research questions defined in Chapter 1 and outline our answers to them based on the studies we conducted. Then we discuss the implications of our work for software developers, tool builders, and researchers. Finally, we sketch opportunities for future work.

7.1 Revisiting Our Research Questions

Let us revisit the five research questions guiding each of our separate studies and summarize the insights we gained during this thesis.

Chapter 2

What factors are relevant to developers working with a developer-centric test amplification tool?

In our first study, we explored an initial design of developer-centric test amplification with the help of a dedicated test exploration tool that facilitates the communication between

the test amplification tool and the software developer. We implemented this design in our plugin *TestCube* for the IntelliJ integrated development environment. With this prototype, we let 16 developers try out our idea of developer-centric test amplification. During our semi-structured interviews with these developers, we observed key factors to make both the amplified tests and the test exploration tool suited for the developer-centric interaction. We also collected which kinds of information the developers sought about amplified tests and what value developer-centric test amplification brings to developers. We learned that the central factor for the developers accepting a test is that they understand a variety of aspects about the test, like its behavior or targeted code under test, and that the tested scenario is relevant for the developers. The test exploration tool should be easy to use and actively manage the developers' expectations about which tests it can generate and how long this will take. The tool should also manage which information is shown to or reachable by the developer, in order to not overwhelm them and let them focus on their current task during the interaction.

We give two recommendations arising from this study. The first recommendation is to consciously design the interaction between the software developer and the automatic test amplification tool. Our second recommendation is that in such a developer-centric use case, the understandability and relevance of an amplified test should be prioritized over its improvement of coverage metrics.

Chapter 3

How should a visualization of the execution behavior and coverage impact of an amplified test be designed to help software developers judge the amplified test?

For our second study, we developed TestImpactGraph, a visualization that shows the methods executed by a test and highlights the additional coverage provided by the test compared to the existing test suite. In our think-aloud study, we learned that the visualization is helpful to developers, and that they want access to a variety of information about the test execution. Exploring the visualization, our participants asked about other tests that cover the same code, which could help them judge the usefulness of the visualized test. Our visualization made deep coverage contributions visible, where the additionally covered lines are several method calls away from the test method. This could be an unintended coverage contribution or point to a lack of more direct unit testing.

From our observations, we conjectured that using a visualization like TestImpactGraph can enable developers to gain more refined insights into the behavior and coverage of a test in comparison to the rest of the test suite. These insights can also be beneficial when reviewing tests contributed by other developers or when refactoring a test suite to be smaller or more modular.

Chapter 4

What changes do developers make to amplified tests before incorporating them into their maintained test suite?

In our third study, we investigated more broadly which edit and selection actions

developers should expect to make when working with amplified tests. To study the realistic opinions of developers who are experts with the systems under test, we amplified tests for popular open source Java projects. We submitted pull requests with tests that improve their test suites, and analyzed the responses of the maintainers who reviewed our pull requests. Being mindful to not antagonize the maintainers with low quality contributions, we had to manually select and edit some of the amplified tests before opening the pull requests.

Based on the checklists we developed for our manual preparation and the changes requested by the pull request reviewers, we formulate general guidelines for developers on how to select and edit amplified tests before including them in a test suite. We observed that these changes fall largely into two main categories. The first one concerns project-specific changes. These could be automated by configuring the test amplification process to fit the specific project, like automatic linter fixes or blacklisting methods that do not require testing. The second category consists of changes that are hard to automate because they highly benefit from the developer understanding the test. As the developers already aim to understand the test before accepting it into their test suite, we conjecture that supporting them in understanding the test and helping them make these changes confidently might be more fruitful than further trying to automate these changes.

Chapter 5

How does developer guidance towards a coverage target impact the developer-centric test amplification process?

In the fourth study presented in this thesis, we investigate the impact of explicit guidance by the developer on the test amplification process and the comprehension of the amplified tests. We extended our IntelliJ plugin developed in Chapter 2 with a control-flow graph visualization that lets developers indicate which branch of a method they want to cover with an amplified test. We then reuse the same visualization to communicate the coverage of the resulting amplified tests to the developer. Our technical study shows that even with simple changes to the test amplification process, the guidance leads to a more focused test amplification that produces a larger proportion of fitting tests. Our user study showed that the active guidance by the developer and the control-flow graph visualizations makes the amplified tests easier to understand. However, the user guidance also creates new requirements, like managing the expectations of which branches can be covered, or a high speed of the test generation so that the response time feels interactive for the user.

Overall, we see that there are trade-offs between the directed and non-directed variations of test amplification. We recommend choosing the directed or non-directed test amplification depending on the use case in which the developer wants to generate tests: When they are looking for tests accompanying the code they are currently writing, directed test amplification is more helpful, while non-directed test amplification is better suited for dedicatedly improving a test suite as a whole.

Chapter 6

What are developer's reactions when proposing fuzzing-based tests that would close coverage gaps after being completed with a functional oracle?

For our fifth study, we collaborated with Mozilla and together explored how we can leverage their established fuzzing tooling to generated tests for code that is not yet covered by their test suite. With a fuzzer we generate a test input that executes a not-yet-tested block of code from the Firefox browser and propose these partial tests to the developers. Our idea was that the developers complete the partial tests into functional tests by adding an explicit oracle to it, i.e., an assertion that checks that the targeted line of code behaves correctly. From the developers' reactions to our proposed tests we learn that the test-worthiness of the targeted coverage gap is a central concern for developers. After designing a filter to exclude the mentioned irrelevant coverage gaps, we discuss a set of remaining coverage gaps that we can generate partial tests for with the developers.

We learn in more depth what aspects make a coverage gap (not) worth closing, such as the component's involvement in recent bugs or the effort to close that coverage gap by writing or completing a test. The developers pointed out the effort of understanding the generated test compared to writing a test from scratch for the coverage gap, which was perceived as easier. They also asked to integrate the test proposals better in their test suite and development workflow.

7.2 How to Design an Effective Collaboration Between Software Developers and Automatic Test Amplification Tools

Through the five studies we conducted for this thesis, we gained an understanding on how to build developer-centric test amplification tools (Chapters 2, 3 and 5), identified which activities developers can expect to perform when working with developer-centric test amplification (Chapters 2 and 4), and recognized that developer involvement comes at a cost and opens new requirements (Chapters 5 and 6). To answer our overarching research question and assemble the insights from our studies, we will now describe how one should design an effective collaboration between software developers and automatic test amplification tools based on our research.

Active and careful design of the communication between developer and automatic tool is central, because developers want to know and understand a plethora of aspects from a test case in order to judge whether they should include a test in their test suite (Chapters 2 to 6). Important here is that developers seek a variety of information depending on their situation, their software project and the developer themselves (Chapters 2 to 4). They could for example seek information about the intended behavior of the test, the contributed coverage, the test's runtime or whether it passes (Chapters 2 and 3). This information should be accessible in findable locations and not be shown all at once (Chapters 2 and 3). Giving direction in the process benefits comprehension, but increases expectations of the developers towards covering targeted branches and interactive speed (Chapter 5). The developer should be supported and encouraged to take action. For example, this action

could be to modify the test based on their insight about the project, the code under test and the test suite. Developers might also extend the test suite or adapt the code under test based on inspiration from the amplified test (Chapters 2, 4 and 6). The developer should be informed about what they can expect when interacting with the test amplification tool, what tests the tool can produce for them, and which actions and contributions are expected from them in this process (Chapters 2 and 4 to 6).

One of the central aspects to consider is the effort-to-value ratio that the tool provides to the developer. The effort requested from the developer must match their perceived value from the resulting tests. This ratio can be adjusted on both sides. For example, one can reduce the effort required from the developer by supporting them in understanding the test (Chapters 2 to 3, 5 and 6), provide as complete as possible tests (Chapter 6), and integrate the tools and processes into their existing workflow (Chapters 2, 5 and 6). On the other side, one can increase the value that developers gain from the interaction, by creating more useful tests for the developers, e.g., by targeting coverage gaps that they find relevant to test (Chapters 2, 4 and 6) or by targeting code that they recently worked on (Chapters 5 and 6). This also concerns the developers' perception and comprehension of the additional value that an amplified test brings, by more effectively communicating the behavior and impact in terms that are meaningful to the developers (Chapters 3 to 6), e.g., by explaining which behavior of their software is tested additionally by an amplified test.

7.3 Implications

In this section, we discuss the implications of our work for software developers and the larger society, for tool builders that aim to help software developers in their work and for researchers studying software testing and the collaboration between software developers and automatic test generation tools.

7.3.1 Implications for Software Developers and Society

Software developers who are interested in improving their test suites can **use our openly available test amplification and exploration tools** to amplify tests in their own software projects and with that improve their test suites. Our available tools are the IntelliJ Plugin *TestCube*[1] and our developer-centric adaption of DSpot[2]. The developers also benefit from our insights about which use cases test amplification is suited for, and what actions and contributions are expected from them when they work with developer-centric test amplification tools. This enables them to **take an informed decision about whether and when to apply developer-centric test amplification** in their software projects or day-to-day development work.

Our insights can potentially also help developers improve their communication with other developers during code reviews that involve tests. Just as with automatic test amplification, reviewers who judge a test that is proposed during a pull request need to understand, judge and propose edits for tests that are supposed to improve a test suite. Therefore, the contributors proposing such tests can leverage our insights to decide what information

[1] `https://plugins.jetbrains.com/plugin/14678-test-cube`
[2] `https://github.com/TestShiftProject/dspot`

about tests they should convey to the maintainers to convince them of the behavior and provided value of their tests.

As our work on test amplification advances one of the technologies helping developers create stronger test suites more easily, we contribute to an improvement of the quality assurance of the software we build and maintain in the future. Better quality assurance helps to avoid bugs and other problems so that our society can better rely on the high quality software that developers create.

7.3.2 Implications for Tool Builders

For tool builders, who design and implement tools that help software developers in their day-to-day work, our research provides guidance on a variety of aspects concerning how to design effective collaborations between software developers and automatic test amplification tools. Beyond that, we clearly saw that **in a developer-centric workflow**, where tools rely on the developers adopting and using a tool, **developer satisfaction is more important than improving metrics** about the quality of a test suite. This means, e.g., that the developer being happy about the value that they receive from interacting and working with a test amplification tool is more relevant to the success of the tool than how much measurable improvement the amplified tests bring to the test suites.

In this consideration, **the value perceived by the developer is leading**. It is not sufficient to only provide high quality and helpful generations to the developers, but we also need to effectively communicate and convince them about the value that is provided by the artifacts we produce. To achieve such a successful communication, we recommend considering the developer interaction of tools separately from the pure generation functionality. For test amplification or generation, tool builders can learn from the insights described in this thesis, but more importantly, they should conduct their own qualitative research to understand how developers interact with their tools. To design this interaction, tool builders should consider the various roles that the developers take on, such as writers, reviewers or future readers and users of the tests or code that the tools create. These roles also point to the tasks and actions developers might perform during the interaction, such as understanding, judging, editing or extending the output generated by tools.

We observed that involving developers in an active role during a collaboration with automatic tools can help tackle otherwise hard to automate challenges. However, we recommend tool builders to carefully **balance the effort asked from the developer and the value perceived by the developer**. This can be done by reducing the effort, e.g., through functionalities supporting developer comprehension, or by increasing the perceived value, e.g., by adapting optimization targets to fit the developer better or investing in more meaningful communication about the beneficial contributions of a tool's output.

7.3.3 Implications for Researchers

A plethora of software engineering research, including the work on test generation and amplification, focuses on further and further automating the tasks of software developers. Nevertheless, even if we succeed in automating some of these tasks, the developers later still have to interact with the artifacts we produce, such as developers using failing tests to localize a fault in the source code or developers evaluating amplified tests before incorporating them into their test suite. As this requires the developers to understand our

generated artifacts, we could already leverage this comprehension earlier by collaborating together with the developers to tackle activities that are hard to fully automate. Therefore, we recommend researchers to **consider collaborating with and actively supporting the developers instead of only focusing on automating** their tasks.

In our studies we observed that the developers find it important to understand the behavior and the value of an automatically amplified test before they include it in their test suite. We conjecture that this **importance of understanding the output of generative tools also applies to the usage of generative artificial intelligence for software engineering tasks.** Our vision of a consciously designed and effective collaboration with software developers, as well as our concrete insights about how to communicate about tests that improve a given test suite, could be applied to improve the interaction of generative artificial intelligence with software developers.

Throughout this thesis, and especially in the last study (Chapter 6), we observed that developers find different parts of their software more or less relevant to test with the automated regression tests that we generate. This is in line with the common recommendation to not aim for 100 % code coverage for a manually written test suite. However, we still lack a formal and empirically validated understanding on what parts of a software system should be tested with regression tests according to the judgement by the software developers. Therefore, we call for the development of a **developer-driven test adequacy metric** that can give more constructive feedback about whether a test suite is adequately testing the software system, based on the trade-offs developers have to make between the effort and value of automatic tests. Such a metric could help guide automatic test generation tools towards coverage goals that are relevant for developers, but also help maintainers and managers to have a clearer understanding whether the right 80 % of their code is covered by tests.

7.4 Future Work

In this section, we sketch opportunities for future work that build upon the insights from this thesis.

Study Developer-Centric Test Amplification in Different Contexts: The studies in this thesis mainly investigate test amplification for Java and single interactions of developers with amplification tools or amplified tests. Future research should investigate how our findings transfer to other contexts. This could be other programming languages and paradigms, such as dynamically typed languages [170, 171], or domain-specific tests, e.g., for distributed systems or machine learning. A second context to investigate is the long-term use of developer-centric test amplification in a software project by one or multiple developers in a longitudinal study [215]. The option to use test amplification might impact their decision making around developing tests, or the long-term experience using test amplification might impact the developer's interaction with and needs from the tool.

Learn From Developer Interactions With Developer-Centric Test Amplification Tools and Amplified Tests: Throughout our studies, we observed that the interaction with and requirements towards test amplification varied depending on the developer, their

use case and the project under test. Future generations of test amplification tools should take this diversity into account and adapt their generated tests to fit the developer's current context. One approach for this is to track the interactions of a user with the tool and the amplified tests, learning from their manual edits to and selections of the tests. Such implicit feedback could be combined with explicit feedback to customize the test amplification process and the interaction with the developer.

Decouple Test Generation From Developer Interaction: Several of the design decisions we took in our work on developer-centric test amplification were based on the requirement to make the test amplification fast enough, so that a developer can interact with it in real-time. This motivated our choice for instruction coverage instead of mutation score in our first design in Chapter 2. When incorporating the developer's guidance towards a certain branch in Chapter 5, we opted for the simple modification of calling the targeted method during the mutations performed by the test amplification. Decoupling the test generation process from the interaction moment would allow us to explore more runtime-consuming test generation approaches while still providing an interactive experience to the developer. For example, we would generate a variety of tests in the background and save them. Then, when the developer requests a test for a certain coverage target, we pick out the most fitting tests and propose these to the developer in the test exploration tool. This decoupling would also enable us to integrate the developer-centric interaction of our proposed test exploration tool with other test generation approaches, like search-based or LLM-based test generation.

Just-in-Time Test Generation: The current test generation approaches generally require the code under test to be already written. However, with today's extensive education about the importance of testing, developers strive to create tests directly when they write new functionality [8]. The next step in developer-centric test generation would be to propose fitting tests directly while the developer is writing their code. We sketched this idea called Just-in-time test generation in a talk at the SMILESENG summer school [216]. We envision a tool that is integrated into the developer's IDE and detects when the developer finished writing a coherent change or new functionality. Then it automatically creates a matching test case in the background and proposes it to the developer, referring to their just finished change. This quick generation is feasible because in the spirit of test amplification the new tests are closely based on the existing tests for the neighboring code, with small changes to test the new functionality. There are many challenges to tackle in order to realize such a just-in-time test generation tool. These include, for example, detecting the completion of a test-worthy change, generating a test that matches the change in the very limited time available, or designing effective messages to propose the test to the developer.

Developer-Driven Test Adequacy Metric: In Chapter 6, we observed that not all code is equally worth testing. This matches with common recommendations to not aim for 100 % code coverage with automatic regression test suites, because it might not be worth the effort that is required to test every possible scenario. The filter we designed in Chapter 6 and our discussions with the developers are a starting point to formalize which code in a software system should be tested with regression tests according to the judgement by

the software developers. A more refined model would be helpful to select which tests to keep after the test amplification, or to use as a more developer-centered fitness function when automatically optimizing test suites. It can also help software engineers to gain a more refined judgement of the test coverage of their current test suites, giving insights into whether the right 80 % of their code is executed by tests. To create such a refined, developer-driven model, we propose to combine empirical studies analyzing which lines of code are covered in mature open source projects [133], with further qualitative interviews and discussions with software developers.

Test Amplification With Large Language Models: Another approach to generate tests is by leveraging machine learning, in particular large language models (LLMs) [19, 217, 218]. LLMs are powerful at producing natural, realistic and readable test data and code. Future work should investigate how LLMs can be leveraged for test amplification, for example, by including original tests from the existing test suite or guidance towards a coverage target in the prompt. LLMs could also work together with algorithmic approaches by contributing natural test data or improving the readability of tests in a post-processing step. Similar to the work in this thesis, the interaction of the developer with an LLM-based test generation tool should be investigated to uncover how usable the generated tests are [219].

Amplification for Non-Automated Tests: Our studies and a majority of automatic generation tools focus on generating automatically executable tests. However, higher in the testing pyramid there are also valuable types of tests that are more expensive to execute. These tests rely on human action, e.g., manual tests or user tests [220]. Future work should explore how the idea of test amplification and our insights on communicating tests transfer to the generation of manual tests. Due to the size of the executed code, one can no longer so closely rely on the code under test to guide the test amplification. In addition, executing tests to validate that they pass or to measure their contribution to the test suite is not as straightforward then.

Incorporate Knowledge From Other Artifacts in the Developer Workflow: In our research, we choose to work towards addressing the relevance problem and the oracle problem in automatic test generation by collaborating with the developer. Another option is to extract the required knowledge from other artifacts in the software engineering process. Just as integrating all information for a task at hand helps developers [221], test amplification tools could leverage information from just-in-time requirements documents like issues [222] to inform the tool which paths to test and which expected outputs to compare to.

Intertwinement With Software Testing Education: In our vision of developer-centric test amplification, we take the developer's judgement as the basis to decide what is a good test. However, the experience, knowledge and awareness of software testing varies from developer to developer [165, 223] possibly due to their varying formal and informal education. At the same time, automatic generation tools like the ones we propose can take a role as an educator by guiding the developer and giving examples of high-quality tests [224]. Future research should explore how the use of developer-centric test amplification tools

differs for software developers with different experience levels and its educational impact on the developers.

Help Developers Write the First Test: Test amplification requires an original, manually-written test to base the amplified tests on. While mature projects frequently have established test suites that we can improve, this does not hold for new or smaller projects. These can therefore hardly benefit from test amplification. From our own experience, writing such a first test of a test suite can be quite difficult. This is also supported by Swillus and Zaidman's theory [117] that the growing complexity in still small software projects both creates the need for tests but also impairs the creation of said test suite. This is why future studies should seek methods to automatically generate initial tests and a test suite for an existing project, or to support the developer in creating them. One approach could be to carve out these tests from system tests or manual whole-system interactions [159]. Then, we can investigate how such generated initial tests used as original tests impact the effectiveness of test amplification. Just as generative machine learning models perform worse when trained with generated data [225], amplifying generated test might present difficulties and require adaptations of the amplification process.

Bibliography

References

[1] James A Whittaker, Jason Arbon, and Jeff Carollo. *How Google Tests Software*. Addison-Wesley, 2012.

[2] Maurício Aniche. *Effective Software Testing: A Developer's Guide*. Simon and Schuster, 2022.

[3] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Pearson Education, 2007.

[4] Alexander Tarlinder. *Developer Testing: Building Quality Into Software*. Addison-Wesley Professional, 2016.

[5] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023.

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 559–562. IEEE CS, 2015.

[7] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 179–190. ACM, 2015.

[8] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: patterns, beliefs, and behavior. *IEEE Trans. Software Eng.*, 45(3):261–284, 2019.

[9] André N. Meyer, Gail C. Murphy, Thomas Fritz, and Thomas Zimmermann. Developers' diverging perceptions of productivity. In *Rethinking Productivity in Software Engineering*, pages 137–146. Apress open / Springer, 2019.

[10] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and European Software Engineering Conference (ESEC)*, pages 416–419. ACM, 2011.

[11] Gordon Fraser and Andrea Arcuri. EvoSuite: On the challenges of test case generation in the real world. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 362–369. IEEE CS, 2013.

[12] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *International Conference on Tests and Proofs (TAP)*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[13] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with DSpot: A study with ten mature open-source projects. *Empir. Softw. Eng.*, 24(4):2603–2635, 2019.

[14] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 75–84. IEEE CS, 2007.

[15] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE, 2023.

[16] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Trans. Software Eng.*, 44(2):122–158, 2018.

[17] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Software Eng.*, 39(2):276–291, 2013.

[18] Benjamin Danglot, Martin Monperrus, Walter Rudametkin, and Benoit Baudry. An approach and benchmark to detect behavioral changes of commits in continuous integration. *Empir. Softw. Eng.*, 25(4):2379–2415, 2020.

[19] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.

[20] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 23—32. IEEE CS, 2011.

[21] STAMP. Use cases validation report v3. `https://github.com/STAMP-project/docs-forum/blob/master/docs/`, 2019.

[22] Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella. Good things come in threes: Improving search-based crash reproduction with helper objectives. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 211–223. IEEE, 2020.

[23] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. Botsing, a search-based crash reproduction framework for Java. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1278–1282. IEEE, 2020.

[24] Fitash Ul Haq, Donghwan Shin, Lionel C. Briand, Thomas Stifter, and Jun Wang. Automatic test suite generation for key-points detection DNNs using many-objective search (experience paper). In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 91–102. ACM, 2021.

[25] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[26] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search-based software engineering with Sapienz at Facebook. In *International Symposium on Search-Based Software Engineering (SSBSE)*, volume 11036 of *LNCS*, pages 3–45. Springer, 2018.

[27] Sergio Segura, Gordon Fraser, Ana Belén Sánchez, and Antonio Ruiz Cortés. A survey on metamorphic testing. *IEEE Trans. Software Eng.*, 42(9):805–824, 2016.

[28] Maurício Aniche, Christoph Treude, and Andy Zaidman. How developers engineer test cases: An observational study. *IEEE Trans. Software Eng.*, 48(12):4925–4946, 2022.

[29] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 107–118. ACM, 2015.

[30] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. An empirical investigation on the readability of manual and generated test cases. In *IEEE International Conference on Program Comprehension (ICPC)*, pages 348–351. ACM, 2018.

[31] Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 625–636. ACM, 2016.

[32] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 57–67. ACM, 2017.

[33] Nienke Nijkamp, Carolin Brandt, and Andy Zaidman. Naming amplified tests based on improved coverage. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 237–241. IEEE, 2021.

[34] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 287–298. IEEE, 2020.

[35] 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 *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 547–558. ACM, 2016.

[36] Simon Bihel and Benoit Baudry. Adapting amplified unit tests for human comprehension. *KTH Internship Report*, 2018.

[37] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P Robillard. Generating unit tests for documentation. *IEEE Trans. Software Eng.*, 2021.

[38] 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.

[39] Daniel Hoffman and Paul Strooper. API documentation with executable examples. *J. Syst. Softw.*, 66(2):143–156, 2003.

[40] Pavneet Singh Kochhar, Xin Xia, and David Lo. Practitioners' views on good software testing practices. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 61–70. IEEE/ACM, 2019.

[41] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.

[42] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *J. Syst. Softw.*, 157, 2019.

[43] Joris Van Geet and Andy Zaidman. A lightweight approach to determining the adequacy of tests as documentation. *Proc. PCODA*, 6:21–26, 2006.

[44] Fabian Trautsch, Steffen Herbold, and Jens Grabowski. Are unit and integration test definitions still valid for modern java projects? An empirical study on open-source projects. *J. Syst. Softw.*, 159, 2020.

[45] Mohammad Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 263–272. IEEE CS, 2017.

[46] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? A controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4):23:1–23:49, 2015.

[47] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 338–349. ACM, 2015.

[48] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2123–2138. ACM, 2018.

[49] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[50] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, 2004.

[51] Paul Ralph, Sebastian Baltes, Domenico Bianculli, Yvonne Dittrich, Michael Felderer, Robert Feldt, Antonio Filieri, Carlo Alberto Furia, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara A. Kitchenham, Romain Robbes, Daniel Méndez, Jefferson Seide Molléri, Diomidis Spinellis, Miroslaw Staron, Klaas-Jan Stol, Damian A. Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, and Sira Vegas. ACM SIGSOFT empirical standards. *CoRR*, abs/2010.03525, 2020.

[52] Carolin Brandt and Andy Zaidman. Developer-centric test amplification. *Empir. Softw. Eng.*, 27(4):96, 2022.

[53] Carolin Brandt and Andy Zaidman. How does this new developer test fit in? A visualization to understand amplified test cases. In *Working Conference on Software Visualization (VISSOFT)*, pages 17–28. IEEE, 2022.

[54] Carolin Brandt, Ali Khatami, Mairieli Wessel, and Andy Zaidman. Shaken, not stirred. How developers like their amplified tests. *IEEE Transactions on Software Engineering*, 50(5):1264–1280, 2024.

[55] Carolin Brandt, Danyao Wang, and Andy Zaidman. When to let the developer guide: Trade-offs between open and guided test amplification. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 231–241. IEEE, 2023.

[56] Carolin Brandt, Marco Castelluccio, Christian Holler, Jason Kratzer, Andy Zaidman, and Alberto Bacchelli. Mind the gap: What working with developers on fuzz tests taught us about coverage gaps. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2024.

[57] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. On the effectiveness of manual and automatic unit test generation: Ten years later. In *International Conference on Mining Software Repositories (MSR)*, pages 121–125. IEEE/ACM, 2019.

[58] Kent L. Beck. *Test-Driven Development - By Example*. Addison-Wesley, 2003.

[59] Davide Spadini, Maurício Finavaro Aniche, Margaret-Anne D. Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review: Why and how developers review tests. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 677–687. ACM, 2018.

[60] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. Automatic test case generation: What if test code quality matters? In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 130–141. ACM, 2016.

[61] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. Pizza versus pinsa: On the perception and measurability of unit test code quality. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 336–347. IEEE, 2020.

[62] Xinhong Liu and Reid Holmes. Exploring developer preferences for visualizing external information within source code editors. In *Working Conference on Software Visualization (VISSOFT)*, pages 27–37. IEEE, 2020.

[63] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empir. Softw. Eng.*, 18(3):594–623, 2013.

[64] Carolin Brandt and Andy Zaidman. Appendix to "Developer-centric test amplification: The interplay between automatic generation and human exploration". `https://doi.org/10.5281/zenodo.5254870`, 2021.

[65] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the System Usability Scale. *Int. J. Hum. Comput. Interact.*, 24(6):574–594, 2008.

[66] Juliet M Corbin and Anselm L Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, 1990.

[67] Yan Zhang and Barbara M Wildemuth. Unstructured interviews. *Applications of social research methods to questions in information and library science*, pages 222–231, 2009.

[68] Nigel Bevan. International standards for HCI and usability. *Int. J. Hum. Comput. Stud.*, 55(4):533–552, 2001.

[69] Marllos Paiva Prado and Auri Marcelo Rizzo Vincenzi. Towards cognitive support for unit testing: A qualitative study with practitioners. *J. Syst. Softw.*, 141:66–84, 2018.

[70] Infinitest. Infinitest - the continuous test runner for the JVM. https://infinitest.github.io/, 2021.

[71] Wessel Oosterbroek, Carolin Brandt, and Andy Zaidman. Removing redundant statements in amplified test cases. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 242–246. IEEE, 2021.

[72] Boyang Li, Christopher Vendome, Mario Linares Vásquez, Denys Poshyvanyk, and Nicholas A. Kraft. Automatically documenting unit test cases. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 341–352. IEEE CS, 2016.

[73] Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. What factors make SQL test cases understandable for testers? A human study of automated test data generation techniques. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 437–448. IEEE, 2019.

[74] Andrea Arcuri, José Campos, and Gordon Fraser. Unit test generation during software development: EvoSuite plugins for Maven, IntelliJ and Jenkins. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–408. IEEE CS, 2016.

[75] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *International Conference on Mining Software Repositories (MSR)*, pages 356–367. IEEE CS, 2017.

[76] STAMP. STAMP project: Eclipse IDE. `https://github.com/STAMP-project/stamp-ide`, 2019.

[77] Bogdan Marculescu, Robert Feldt, and Richard Torkar. A concept for an interactive search-based software testing system. In *International Symposium on Search-Based Software Engineering (SSBSE)*, volume 7515 of *LNCS*, pages 273–278. Springer, 2012.

[78] Bogdan Marculescu, Robert Feldt, Richard Torkar, and Simon M. Poulding. Transferring interactive search-based software testing to industry. *J. Syst. Softw.*, 142:156–170, 2018.

[79] K. Anders Ericsson and Herbert A. Simon. How to study thinking in everyday life: Contrasting think-aloud protocols with descriptions and explanations of thinking. *Mind, Culture, and Activity*, 5(3):178–186, 1998.

[80] Panagiotis K. Linos, Philippe Aubet, Laurent Dumas, Yan Helleboid, Patricia Lejeune, and Philippe Tulula. Facilitating the comprehension of C-programs: An experimental study. In *IEEE Workshop on Program Comprehension (WPC)*, pages 55–63. IEEE, 1993.

[81] Holger M. Kienle and Hausi A. Müller. Requirements of software visualization tools: A literature survey. In *IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 2–9. IEEE CS, 2007.

[82] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the elipse IDE? *IEEE Softw.*, 23(4):76–83, 2006.

[83] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Software Eng.*, (12):971–987, 2006.

[84] Michael Desmond, Margaret-Anne D. Storey, and Chris Exton. Fluid source code views. In *International Conference on Program Comprehension (ICPC)*, pages 260–263. IEEE CS, 2006.

[85] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan O. Borchers. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Annual ACM Symposium on User Interface Software and Technology*, pages 217–224. ACM, 2011.

[86] Andrew Bragdon, Robert C. Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *International Conference on Human Factors in Computing Systems (CHI)*, pages 2503–2512. ACM, 2010.

[87] Andrew Bragdon, Steven P. Reiss, Robert C. Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 455–464. ACM, 2010.

[88] J. Lawrance, Steven Clarke, Margaret Burnett, and Gregg Rothermel. How well do professional developers test with code coverage visualizations? An empirical study. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 53–60. IEEE, 2005.

[89] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages (VL)*, pages 336–343. IEEE, 1996.

[90] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *J. Syst. Softw.*, 44(3):171–185, 1999.

[91] Felice Salviulo and Giuseppe Scanniello. Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 48:1–48:10. ACM, 2014.

[92] Chak Shun Yu, Christoph Treude, and Maurício Finavaro Aniche. Comprehending test code: An empirical study. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 501–512. IEEE, 2019.

[93] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 209–218. IEEE CS, 2009.

[94] Victor Hurdugaci and Andy Zaidman. Aiding software developers to maintain developer tests. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 11–20. IEEE CS, 2012.

[95] Dabo Sun and Kenny Wong. On evaluating the layout of UML class diagrams for program comprehension. In *International Workshop on Program Comprehension (IWPC)*, pages 317–326. IEEE CS, 2005.

[96] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. The aesthetics of graph visualization. In *International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging*, pages 57–64. Eurographics Association, 2007.

[97] Michael Hilton, Jonathan Bell, and Darko Marinov. A large-scale study of test coverage evolution. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 53–63. ACM, 2018.

[98] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. Presentation abstract: Generating class integration tests using call site information. In *Belgium-Netherlands Software Evolution Workshop (BENEVOL)*, 2019.

[99] Mark Grechanik and Gurudev Devanla. Generating integration tests automatically using frequent patterns of method execution sequences. In *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 209–280. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019.

[100] Mauro Pezzè, Konstantin Rubinov, and Jochen Wuttke. Generating effective integration test cases from unit ones. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 11–20. IEEE CS, 2013.

[101] Markus Borg, Andreas Brytting, and Daniel Hansson. An analytical view of test results using cityscapes. In *Design and Verification Conference and Exhibition United States (DVCON US)*, 2018.

[102] Gergõ Balogh, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. Using the city metaphor for visualizing test-related metrics. In *International Workshop on Validating Software Tests (VST@SANER)*, pages 17–20. IEEE CS, 2016.

[103] Michael Perscheid, Damien Cassou, and Robert Hirschfeld. Test quality feedback improving effectivity and efficiency of unit testing. In *International Conference on Creating, Connecting and Collaborating through Computing*, pages 60–67. IEEE, 2012.

[104] Rudolfs Opmanis, Paulis Kikusts, and Martins Opmanis. Visualization of large-scale application testing results. *Baltic Journal of Modern Computing*, 4(1):34, 2016.

[105] Manuel Breugelmans and Bart Van Rompaey. TestQ: Exploring structural and maintenance characteristics of unit test suites. In *International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)*. Citeseer, 2008.

[106] Vanessa Peña Araya. Test blueprint: An effective visual support for test coverage. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1140–1142. ACM, 2011.

[107] Ani Rahmani, Joe Lian Min, and Asri Maspupah. An evaluation of code coverage adequacy in automatic testing using control flow graph visualization. In *IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, pages 239–244. IEEE, 2020.

[108] Bas Cornelissen, Arie van Deursen, Leon Moonen, and Andy Zaidman. Visualizing testsuites to aid in software understanding. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 213–222. IEEE, 2007.

[109] Arthur-Jozsef Molnar. Live visualization of GUI application code coverage with GUITracer. *CoRR*, abs/1702.08013, 2017.

[110] Paul V. Gestwicki and Bharat Jayaraman. Interactive visualization of Java programs. In *IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC)*, pages 226–235. IEEE CS, 2002.

[111] Philipp Bouillon, Jens Krinke, Nils Meyer, and Friedrich Steimann. EzUnit: A framework for associating failed unit tests with potential programming errors. In *International Conference on Agile Processes in Software Engineering and Extreme Programming (XP)*, volume 4536 of *LNCS*, pages 101–104. Springer, 2007.

[112] Nadera Aljawabrah, Tamás Gergely, Sanjay Misra, and Luis Fernández Sanz. Automated recovery and visualization of test-to-code traceability (TCT) links: An evaluation. *IEEE Access*, 9:40111–40123, 2021.

[113] Nadera Aljawabrah, Tamás Gergely, and Mohammad Kharabsheh. Understanding test-to-code traceability links: The need for a better visualizing model. In *International Conference on Computational Science and Its Applications (ICCSA)*, volume 11622 of *LNCS*, pages 428–441. Springer, 2019.

[114] Andreina Cota Vidaure, Evelyn Cusi Lopez, Juan Pablo Sandoval Alcocer, and Alexandre Bergel. TestEvoViz: Visual introspection for genetically-based test coverage evolution. In *Working Conference on Software Visualization (VISSOFT)*, pages 1–11. IEEE CS, 2020.

[115] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Software Eng.*, 36(6):742–762, 2010.

[116] Luciano Baresi and Matteo Miraz. TestFul: Automatic unit-test generation for Java classes. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 281–284. ACM, 2010.

[117] Mark Swillus and Andy Zaidman. Sentiment overflow in the testing stack: Analysing software testing posts on stack overflow. *J. Syst. Softw.*, 205:111804, 2023.

[118] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Softw. Pract. Exp.*, 34(11):1025–1050, 2004.

[119] Chris Brown and Chris Parnin. Sorry to bother you: Designing bots for effective recommendations. In *International Workshop on Bots in Software Engineering (BotSE)*, pages 54–58. IEEE/ACM, 2019.

[120] Ali Khatami and Andy Zaidman. State-of-the-practice in quality assurance in open source software development—replication package, 2022.

[121] Roderick Bloem, Robert Koenighofer, Franz Röck, and Michael Tautschnig. Automating test-suite augmentation. In *International Conference on Quality Software*, pages 67–72. IEEE, 2014.

[122] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 257–266. ACM, 2010.

[123] Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. An empirical comparison of EvoSuite and DSpot for improving developer-written test suites with respect to mutation score. In *International Symposium on Search-Based Software Engineering (SSBSE)*, volume 13711 of *LNCS*, pages 19–34. Springer, 2022.

[124] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. The human side of fuzzing: Challenges faced by developers during fuzzing activities. *ACM Trans. Softw. Eng. Methodol.*, 33(1):1–26, nov 2023.

[125] Stephan Plöger, Mischa Meier, and Matthew Smith. A usability evaluation of AFL and libfuzzer with CS students. In *Conference on Human Factors in Computing Systems (CHI)*, pages 186:1–186:18. ACM, 2023.

[126] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. Utopia: Automatic generation of fuzz driver using unit tests. In *IEEE Symposium on Security and Privacy (SP)*, pages 2676–2692. IEEE, 2023.

[127] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1224–1228. IEEE, 2020.

[128] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[129] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. An exploratory study on exception handling bugs in Java programs. *J. Syst. Softw.*, 106:82–101, 2015.

[130] Saurabh Sinha and Mary Jean Harrold. Criteria for testing exception-handling constructs in Java programs. In *International Conference on Software Maintenance (ICSM)*, page 265. IEEE CS, 1999.

[131] Carolin Brandt. Replication package for "Shaken, not stirred. How developers like their amplified tests". https://doi.org/10.5281/zenodo.7034924, 2023.

[132] Ravie Lakshmanan. Minnesota university apologizes for contributing malicious code to the linux project.

[133] Ali Khatami and Andy Zaidman. State-of-the-practice in quality assurance in Java-based open source software development. *Softw. Pract. Exp.*, 2024.

[134] D Randy Garrison, Martha Cleveland-Innes, Marguerite Koole, and James Kappelman. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education*, 9(1):1–8, 2006.

[135] Anselm L Strauss and J. M. Corbin. Basics of qualitative research: Techniques and procedures for developing grounded theory. *SAGE Publications*, 1998.

[136] Barney G Glaser and Anselm L Strauss. *Discovery of Grounded Theory: Strategies for Qualitative Research*. Routledge, 2017.

[137] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 560–564. IEEE, 2021.

[138] 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 *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 358–368. IEEE CS, 2015.

[139] Mika V. Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Trans. Software Eng.*, 35(3):430–448, 2009.

[140] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Jürgens. Modern code reviews in open-source projects: Which problems do they fix? In *Working Conference on Mining Software Repositories (MSR)*, pages 202–211. ACM, 2014.

[141] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE CS, 2013.

[142] Mechelle Gittens, Keri Romanufa, David Godwin, and Jason Racicot. All code coverage is not created equal: A case study in prioritized code coverage. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 131–145, USA, 2006. IBM.

[143] Brian Marick, John Smith, and Mark Jones. How to misuse code coverage. In *International Conference on Testing Computer Software*, pages 16–18, 1999.

[144] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, pages 92–95, 2001.

[145] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. How much does unused code matter for maintenance? In *International Conference on Software Engineering (ICSE)*, pages 1102–1111. IEEE CS, 2012.

[146] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Test smells 20 years later: Detectability, validity, and reliability. *Empir. Softw. Eng.*, 27(7):170, 2022.

[147] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 52–63. ACM, 2014.

[148] Sebastian G. Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Matthew Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Trans. Software Eng.*, 35(1):29–45, 2009.

[149] Salma Messaoudi, Donghwan Shin, Annibale Panichella, Domenico Bianculli, and Lionel C. Briand. Log-based slicing for system-level test cases. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 517–528. ACM, 2021.

[150] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 214–224. ACM, 2015.

[151] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Trans. Reliab.*, 66(4):1213–1228, 2017.

[152] Geraldine Galindo-Gutierrez, Maximiliano Narea Carvajal, Alison Fernandez Blanco, Nicolas Anquetil, and Juan Pablo Sandoval Alcocer. A manual categorization of new quality issues on automatically-generated tests. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023.

[153] David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 90–99. IEEE CS, 2011.

[154] Matt Staats, Michael W. Whalen, and Mats Per Erik Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *International Conference on Software Engineering (ICSE)*, pages 391–400. ACM, 2011.

[155] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering (ICSE)*, pages 402–411. ACM, 2005.

[156] Wei Ma, Thomas Laurent, Milos Ojdanic, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. Commit-aware mutation testing. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 394–405. IEEE, 2020.

[157] Bin Lin, Csaba Nagy, Gabriele Bavota, Andrian Marcus, and Michele Lanza. On the quality of identifiers in test code. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 204–215. IEEE, 2019.

[158] Dietmar Winkler, Pirmin Urbanke, and Rudolf Ramler. What do we know about readability of test code? - A systematic mapping study. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1167–1174. IEEE, 2022.

[159] Amirhossein Deljouyi and Andy Zaidman. Generating understandable unit tests through end-to-end test scenario carving. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 107–118. IEEE, 2023.

[160] Pedro Delgado-Pérez, Aurora Ramírez, Kevin J. Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. InterEvo-TR: Interactive evolutionary test generation with readability assessment. *IEEE Trans. Software Eng.*, 49(4):2580–2596, 2023.

[161] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. CAT-LM training language models on aligned code and tests. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 409–420. IEEE, 2023.

[162] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 181–190. ACM, 2018.

[163] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects. *Empir. Softw. Eng.*, 24(6):3871–3903, 2019.

[164] Lucas Zamprogno, Braxton Hall, Reid Holmes, and Joanne M. Atlee. Dynamic human-in-the-loop assertion generation. *IEEE Trans. Software Eng.*, 49(4):2337–2351, 2023.

[165] Ali Khatami and Andy Zaidman. Quality assurance awareness in open source software projects on GitHub. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 174–185. IEEE, 2023.

[166] Mariam Guizani, Amreeta Chatterjee, Bianca Trinkenreich, Mary Evelyn May, Geraldine J. Noa-Guevara, Liam James Russell, Griselda G. Cuevas Zambrano, Daniel Izquierdo-Cortazar, Igor Steinmacher, Marco Aurélio Gerosa, and Anita Sarma. The long road ahead: Ongoing challenges in contributing to large OSS organizations and what to do. *Proc. ACM Hum. Comput. Interact.*, 5(CSCW2):407:1–407:30, 2021.

[167] Zheying Zhang, Outi Sievi-Korte, Ulla-Talvikki Virta, Hannu-Matti Järvinen, and Davide Taibi. An investigation on the availability of contribution information in open-source projects. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 86–90. IEEE, 2021.

[168] Omar Elazhary, Margaret-Anne D. Storey, Neil A. Ernst, and Andy Zaidman. Do as I do, not as I say: Do contribution guidelines match the github contribution process? In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 286–290. IEEE, 2019.

[169] Andrea Arcuri. An experience report on applying software testing academic results in industry: We need usable automated test generation. *Empir. Softw. Eng.*, 23(4):1959–1981, 2018.

[170] Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel. Small-amp: Test amplification in a dynamically typed language. *Empir. Softw. Eng.*, 27(6):128, 2022.

[171] Ebert Schoofs, Mehrdad Abdi, and Serge Demeyer. AmPyfier: Test amplification in python. *J. Softw. Evol. Process.*, 34(11), 2022.

[172] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In *Software Evolution*, pages 173–202. Springer, 2008.

[173] Aurora Ramírez, José Raúl Romero, and Christopher L. Simons. A systematic review of interaction in search-based software engineering. *IEEE Trans. Software Eng.*, 45(8):760–781, 2019.

[174] Aurora Ramírez, Pedro Delgado-Pérez, Kevin J. Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. Interactivity in the generation of test cases with evolutionary computation. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 2395–2402. IEEE, 2021.

[175] Manabu Kamimura and Gail C. Murphy. Towards generating human-oriented summaries of unit test cases. In *IEEE International Conference on Program Comprehension (ICPC)*, pages 215–218. IEEE CS, 2013.

[176] Daniel Gaston and James Clause. A method for finding missing unit tests. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 92–103. IEEE, 2020.

[177] Chris Parnin and Spencer Rugaber. Resumption strategies for interrupted programming tasks. *Softw. Qual. J.*, 19(1):5–34, Aug 2010.

[178] Anonymous. Online appendix for "When to let the developer guide: Trade-offs between open and guided test amplification". https://doi.org/10.5281/zenodo.8074647, June 2023.

[179] Adrian Santos, Sira Vegas, Oscar Dieste, Fernando Uyaguari, Ayse Tosun, Davide Fucci, Burak Turhan, Giuseppe Scanniello, Simone Romano, Itir Karac, Marco Kuhrmann, Vladimir Mandic, Robert Ramac, Dietmar Pfahl, Christian Engblom, Jarno Kyykka, Kerli Rungi, Carolina Palomeque, Jaroslav Spisak, Markku Oivo, and Natalia Juristo. A family of experiments on test-driven development. *Empir. Softw. Eng.*, 26(3):42, 2021.

[180] Everton da S. Maldonado and Emad Shihab. Detecting and quantifying different types of self-admitted technical debt. In *IEEE International Workshop on Managing Technical Debt (MTD)*, pages 9–15. IEEE CS, 2015.

[181] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Softw.*, 29(6):18–21, 2012.

[182] Zadia Codabux and Byron J. Williams. Managing technical debt: An industrial case study. In *International Workshop on Managing Technical Debt (MTD)*, pages 8–15. IEEE CS, 2013.

[183] Ganesh Samarthyam, Mahesh Muralidharan, and Raghu Kalyan Anna. Understanding test debt. *Trends in Software Testing*, pages 1–17, 2017.

[184] Koushik Sen. Concolic testing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 571–572. ACM, 2007.

[185] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *NASA Formal Methods Symposium (NFM)*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, pages 121–125, 2009.

[186] Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *International Conference on Software Engineering (ICSE)*, pages 132–141. IEEE CS, 2013.

[187] Pouria Derakhshanfar, Xavier Devroey, and Andy Zaidman. Basic block coverage for search-based unit testing and crash reproduction. *Empir. Softw. Eng.*, 27(7):192, 2022.

[188] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, 2013.

[189] Kiran Lakhotia, Mark Harman, and Hamilton Gross. AUSTIN: an open source tool for search based software testing of C programs. *Inf. Softw. Technol.*, 55(1):112–125, 2013.

[190] Josie Holmes, Iftekhar Ahmed, Caius Brindescu, Rahul Gopinath, He Zhang, and Alex Groce. Using relative lines of code to guide automated test generation for python. *CoRR*, abs/2103.07006, 2021.

[191] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search-based software testing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE CS, 2015.

[192] Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. Single-objective versus multi-objectivized optimization for evolutionary crash reproduction. In *International Symposium on Search-Based Software Engineering (SSBSE)*, volume 11036 of *LNCS*, pages 325–340. Springer, 2018.

[193] Zhihong Xu, Myra B. Cohen, and Gregg Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1365–1372. ACM, 2010.

[194] Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. A hybrid directed test suite augmentation technique. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 150–159. IEEE CS, 2011.

[195] Zhihong Xu, Yunho Kim, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. Directed test suite augmentation: An empirical investigation. *Softw. Test. Verification Reliab.*, 25(2):77–114, 2015.

[196] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *International Symposium on Static Analysis (SAS)*, volume 6887 of *LNCS*, pages 95–111. Springer, 2011.

[197] Peter Dinges and Gul A. Agha. Targeted test input generation using symbolic-concrete backward execution. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 31–36. ACM, 2014.

[198] Aidan Murphy, Thomas Laurent, and Anthony Ventresque. The case for grammatical evolution in test generation. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1946–1947. ACM, 2022.

[199] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Workshop on the Future of Software Engineering (FOSE)*, pages 85–103. IEEE CS, 2007.

[200] Matteo Brunetto, Giovanni Denaro, Leonardo Mariani, and Mauro Pezzè. On introducing automatic test case generation in practice: A success story and lessons learned. *J. Syst. Softw.*, 176:110933, 2021.

[201] Domagoj Babic. Sundew: Systematic automated security testing (keynote). In *ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, page 10. ACM, 2017.

[202] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[203] Magiel Bruntink and Arie van Deursen. Predicting class testability using object-oriented metrics. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 136–145. IEEE CS, 2004.

[204] Mozilla Documentation. Mochitest.

[205] Wen Xu, Soyeon Park, and Taesoo Kim. FREEDOM: engineering a state-of-the-art DOM fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 971–986. ACM, 2020.

[206] Google Project Zero. The great dom fuzz-off of 2017.

[207] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[208] Bruce Hanington and Bella Martin. Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions. page 208, 2012.

[209] Gul Calikli and Ayse Bener. Empirical analysis of factors affecting confirmation bias levels of software engineers. *Softw. Qual. J.*, 23(4):695–722, 2015.

[210] Christian R. Prause, Jürgen Werner, Kay Hornig, Sascha Bosecker, and Marco Kuhrmann. Is 100% test coverage a reasonable requirement? lessons learned from a space software project. In *International Conference on Product-Focused Software Process Improvement (PROFES)*, volume 10611 of *LNCS*, pages 351–367. Springer, 2017.

[211] Ali Mesbah, Arie van Deursen, and Danny Roest. Invariant-based automatic testing of modern web applications. *IEEE Trans. Software Eng.*, 38(1):35–53, 2012.

[212] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 201–211. IEEE CS, 2014.

[213] Tao Xie, Jonathan de Halleux, Nikolai Tillmann, and Wolfram Schulte. Teaching and training developer-testing techniques and tool support. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Companion*, pages 175–182. ACM, 2010.

[214] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. White-box fuzzing RPC-based APIs with EvoMaster: An industrial case study. *CoRR*, abs/2208.12743, 2022.

[215] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons, 2012.

[216] Carolin Brandt. Incremental just-in-time test generation in lock-step with code development. In *International Summer School on Search- and Machine Learning-Based Software Engineering (SMILESENG)*, page 35, 2022.

[217] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing. *CoRR*, abs/2308.16557, 2023.

[218] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans. Software Eng.*, 50(1):85–105, 2024.

[219] Khalid El Haji, Carolin Brandt, and Andy Zaidman. Using github copilot for test generation in python: An empirical study. In *International Conference on Automation of Software Test (AST)*, 2024.

[220] Roman Haas, Daniel Elsner, Elmar Jürgens, Alexander Pretschner, and Sven Apel. How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies. In *ACM Joint European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 1281–1291. ACM, 2021.

[221] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 1–11. ACM, 2006.

[222] Neil A. Ernst and Gail C. Murphy. Case studies in just-in-time requirements analysis. In *IEEE International Workshop on Empirical Requirements Engineering (EmpiRE)*, pages 25–32. IEEE CS, 2012.

[223] Baris Ardiç and Andy Zaidman. Hey teachers, teach those kids some software testing. In *IEEE/ACM International Workshop on Software Engineering Education for the Next Generation (SEENG@ICSE)*, pages 9–16. IEEE, 2023.

[224] Cristian-Alexandru Botocan, Piyush Deshmukh, Pavlos Makridis, Jorge Romeu Huidobro, Mathanrajan Sundarrajan, Maurício Aniche, and Andy Zaidman. TestKnight: An interactive assistant to stimulate test engineering. In *IEEE/ACM International Conference on Software Engineering (ICSE) Companion*, pages 222–226. ACM/IEEE, 2022.

[225] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross J. Anderson. The curse of recursion: Training on generated data makes models forget. *CoRR*, abs/2305.17493, 2023.

[226] Carolin Brandt and Andy Zaidman. Strategies and challenges in recruiting interview participants for a qualitative evaluation. In *International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES)*, 2022.

[227] Casper Boone, Carolin E. Brandt, and Andy Zaidman. Fixing continuous integration tests from within the IDE with contextual information. In *IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 287–297. ACM, 2022.

[228] Carolin E. Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. LogChunks: A data set for build log analysis. In *International Conference on Mining Software Repositories (MSR)*, pages 583–587. ACM, 2020.

[229] Monika Pichlmair, Carolin Brandt, Marcel Henrich, Alexander Biederer, Ilhan Aslan, Björn Bittner, and Elisabeth André. Pen-pen: A wellbeing design to help commuters rest and relax. In *Workshop on Human-Habitat for Health (H3): Human-Habitat Multimodal Interaction for Promoting Health and Well-Being in the Internet of Things Era*, pages 1–9, 2018.

Curriculum Vitæ

Carolin Elisabeth Brandt

Date of birth in Munich, Germany *20.08.1996*

Education

PhD Student *2020 - 2024*
Delft University of Technology, Software Engineering Research Group

Visiting PhD Student *August - November 2023*
Hamburg University, Applied Software Technology Group

Master

How to Analyze Build Logs — A Comparative Study of Chunk Retrieval Techniques *2019*
Paper-Based Master Thesis, Guest at the Software Engineering Research Group, Delft University of Technology

Elite Graduate Program Software Engineering *2017 - 2019*
University of Augsburg, Technical University of Munich, Ludwigs-Maximilans-University Munich *Avg. grade: 1.09, 'very good'*

Bachelor

A Description Language for Structural Smells *2017*
Bachelor Thesis in Computer Science

Scholarship "Deutschlandstipendium" *2016-2018*
Combined public and industrial scholarship for excellent students

Admission to best.in.tum *2015*
Group of the best 2% of students at the faculty of computer science

Bachelor Computer Science: Games Engineering *2014 – 2017*
Technical University of Munich *Avg. grade: 1.3, 'very good'*

Secondary School

General Qualification for University Entrance *2006 – 2014*
Werner-Heisenberg-Gymnasium Garching *Avg. grade: 1.3, 'very good'*

Prizes

2023 **Distinguished Reviewer Award**, International Conference on Mining Software Repositories
2022 **Best "New Ideas and Work in Progress" Paper / Presentation**, SMILESENG Summer School
2022 **Award for Best Artifact**, IEEE Working Conference on Software Visualization
2022 **Second Place at the Poster Competition**, Alice & Eve 2022
2021 **International Women in Technology Scholarship**, Personal Scholarship by Zonta
2017 **First Place at the HackaTUM Hackathon**, Category "Bike-Rental Systems Analysis"
2014 **Best Graduate in Chemistry at Werner Heisenberg Gymnasium Garching**, German Chemical Society
2014 **Third Place at the Individual Competition of the "Day of Mathematics"**, University of Ulm

Academic Service

2024 **VST**, Program Committee
2023+24 **MSR**, Junior Program Committee (2023), Program Committee (2024)
2023+24 **MSR: Data and Tool Showcase**, Program Committee
2023 **ACM Transactions on Software Engineering and Methodology**, Journal Reviewer
2022 **FUZZING**, Artifact Evaluation Committee
2022+23 **IEEE Transactions on Software Engineering**, Journal Reviewer
2022 **Journal of Systems and Software**, Journal Reviewer
2022 **BENEVOL, TestVis**, Program Committee
2022 **ECOOP**, Publicity Chair
2022 **Alice & Eve**, Web Chair
2021 **Journal of Software: Evolution and Process**, Journal Reviewer
2021 **VISSOFT**, Program Committee, Publicity Chair
2021 **ISSTA**, Artifact Evaluation Committee
2021 **ICSE**, Organization Committee, Watch Parties and Gather.Town
2020 **ICSE**, Student Volunteer, Streaming in the Europe Timeband
2020 **ESEC/FSE, ASE**, Student Volunteer

List of Publications

11. Carolin Brandt, Ali Khatami, Mairieli Wessel, and Andy Zaidman. Shaken, not stirred. How developers like their amplified tests. *IEEE Transactions on Software Engineering*, 50(5):1264–1280, 2024.

10. Carolin Brandt, Marco Castelluccio, Christian Holler, Jason Kratzer, Andy Zaidman, and Alberto Bacchelli. Mind the gap: What working with developers on fuzz tests taught us about coverage gaps. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2024.

9. Khalid El Haji, Carolin Brandt, and Andy Zaidman. Using github copilot for test generation in python: An empirical study. In *International Conference on Automation of Software Test (AST)*, 2024.

8. Carolin Brandt, Danyao Wang, and Andy Zaidman. When to let the developer guide: Trade-offs between open and guided test amplification. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 231–241. IEEE, 2023.

7. Carolin Brandt and Andy Zaidman. How does this new developer test fit in? A visualization to understand amplified test cases. In *Working Conference on Software Visualization (VISSOFT)*, pages 17–28. IEEE, 2022.

6. Carolin Brandt and Andy Zaidman. Developer-centric test amplification. *Empir. Softw. Eng.*, 27(4):96, 2022.

5. Carolin Brandt and Andy Zaidman. Strategies and challenges in recruiting interview participants for a qualitative evaluation. In *International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES)*, 2022.

4. Carolin Brandt. Incremental just-in-time test generation in lock-step with code development. In *International Summer School on Search- and Machine Learning-Based Software Engineering (SMILESENG)*, page 35, 2022.

5. Casper Boone, Carolin E. Brandt, and Andy Zaidman. Fixing continuous integration tests from within the IDE with contextual information. In *IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 287–297. ACM, 2022.

4. Nienke Nijkamp, Carolin Brandt, and Andy Zaidman. Naming amplified tests based on improved coverage. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 237–241. IEEE, 2021.

3. Wessel Oosterbroek, Carolin Brandt, and Andy Zaidman. Removing redundant statements in amplified test cases. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 242–246. IEEE, 2021.

2. Carolin E. Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. LogChunks: A data set for build log analysis. In *International Conference on Mining Software Repositories (MSR)*, pages 583–587. ACM, 2020.

1. Monika Pichlmair, Carolin Brandt, Marcel Henrich, Alexander Biederer, Ilhan Aslan, Björn Bittner, and Elisabeth André. Pen-pen: A wellbeing design to help commuters rest and relax. In *Workshop on Human-Habitat for Health (H3): Human-Habitat Multimodal Interaction for Promoting Health and Well-Being in the Internet of Things Era*, pages 1–9, 2018.

Included in this thesis.
Won a best paper, tool demonstration, or proposal award.

Titles in the IPA Dissertation Series since 2021

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical.* Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition.* Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05