Supporting Quality In Test Code For Higher Quality Software Systems

Spadini, D.

DOI
10.4233/uuid:fed36b88-bd13-47a8-ad0c-add1e0575f7a
10.5281/zenodo.4600817

Publication date
2021

Document Version
Final published version

Citation (APA)
Spadini, D. (2021). *Supporting Quality In Test Code For Higher Quality Software Systems*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:fed36b88-bd13-47a8-ad0c-add1e0575f7a

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

Supporting Quality In Test Code For Higher Quality Software Systems

Supporting Quality In Test Code For Higher Quality Software Systems

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology,
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates,
to be defended publicly on
Thursday 18 March 2020 at 15.00 o' clock

by

Davide SPADINI

Master of Science in Computer Science,
University of Trento, Italy,
born in Verona, Italy.

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Prof. dr. A. van Deursen,	Technische Universiteit Delft, promotor
Prof. dr. A. Bacchelli,	University of Zurich, promotor
Dr. M. Bruntink,	Software Improvement Group, industry advisor
Independent members:	
Prof. dr. A. Hanjalic,	Technische Universiteit Delft
Prof. dr. D. Spinellis,	Athens University of Economics and Business
Prof. dr. ir. J. Visser,	Leiden University
Prof. dr. J. M. G. Barahona,	Universidad Rey Juan Carlos

The work in the thesis has been carried out under the support of the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 642954.

SWISS NATIONAL SCIENCE FOUNDATION

Keywords: Software Testing, Modern Code Review

Printed by: Gildeprint (https://www.gildeprint.nl/)

Cover: Mirco Spadini

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. The thesis cover is partially derived from a template by Vecteezy.com

It is the very people who no one imagines anything of,
who do the things that no one can imagine

Alan Turing

Contents

Summary

Automated testing has become an essential process for improving the quality of software systems. Automated tests can help ensure that production code is robust under many usage conditions and that code meets performance and security needs. Nevertheless, writing effective tests is challenging and, unfortunately, often neglected. In the first part of this dissertation, we summarize and explain why test and production code are not treated with the same care, and we set our goal: we want to discover new techniques and tools to support developers when writing and reviewing test code. To this aim, we investigate the impact of test design issues on code quality and the practices when writing and reviewing test code.

First, we create and make publicly available Pydriller, a Python framework to analyze software repositories that will help us gather important information for the following studies. Then, we study test design issues and their impact on the overall software code quality, demonstrating how important it is to have a good and effective test suite. Afterward, together with SIG, a company setting in which part of this dissertation was conducted, we study how developers in industry react to these test design issues. Our results show that the current detection rules for test issues are not precise enough and, more importantly, do not support prioritization. We present new rules that can be used to prioritize these issues and show that the results achieved with the new rules better align with developers' perception of importance.

In the second part of the dissertation we focus on how to help developers better reviewing test code. First, we investigate developers' needs when it comes to code reviewing, identifying seven high-level information needs that could be addressed through automated tools, saving up time for reviewers. Then, we focus on code review of test code specifically: we first study when and how developers review test code, identifying current practices, revealing the challenges faced during test code reviews, and uncovering needs for tools that can support the review of test code. Later, we investigate the impact of Test-Driven Code Review (TDR) on the code review effectiveness, showing that it can increase the number of test code issues found. We discuss when TDR can and can not be applied and why not all developers see TDR as a worthy practice.

Samenvatting

Geautomatiseerd testen is een essentieel onderdeel geworden voor het verbeteren van de kwaliteit van een softwaresysteem. Geautomatiseerde testen kunnen helpen om ervoor te zorgen dat productiecode robuust genoeg is om te draaien in de meeste situaties. Daarnaast, helpen deze geautomatiseerde testen ook om de prestatie en veiligheid doelen van een code project te halen. Het schrijven van effectieve testen is alleen niet makkelijk. Ondanks de voordelen van testen, wordt deze stap vaak overgeslagen of onderschat. In het eerste deel van dit proefschrift geven wij een samenvatting en leggen wij uit waarom test en productiecode niet dezelfde aandacht krijgen. Daarnaast zetten wij ook het doel voor dit proefschrift uit: wij willen nieuwe technieken en tools ontdekken om softwareontwikkelaars te kunnen ondersteunen met het schrijven en beoordelen van test code. Om dit te bereiken, onderzoeken we de impact van problemen rondom test ontwerp op de kwaliteit van code en de gewoontes voor het schrijven en beoordelen van test code.

Als eerste, creëren wij Pydriller, een Python framework om software repositories te analyseren dat ons helpt om informatie te verzamelen voor het resterende onderzoek. Hierna, bestuderen wij test ontwerp problemen en de impact die deze kunnen hebben op de kwaliteit van softwarecode. Dit laat zien hoe belangrijk het is om een goede en effectieve test suites te hebben. Na de hand bestuderen wij hoe softwareontwikkelaars in industrie reageren op de naar boven gekomen problemen. Dit doen wij samen met SIG, een bedrijf waar een deel van dit onderzoek heeft plaatsgevonden. Onze resultaten laten zien dat de huidige regels voor het detecteren van problemen in testen niet precies genoeg zijn, en nog belangrijker, geen voorrang volgorde ondersteunen. Wij presenteren nieuwe regels die gebruikt kunnen worden om de gevonden problemen te prioriteren. De gehaalde resultaten laten zien dat deze nieuwe regels beter overeenkomen met wat softwareontwikkelaars belangrijk vinden.

In het tweede deel van dit proefschrift focussen wij op hoe we softwareontwikkelaars beter kunnen helpen om test code te kunnen beoordelen. Als eerste, onderzoeken we de belangrijkste informatie die softwareontwikkelaars nodig hebben voor het beoordelen van code. Hieruit kwamen op een hoog niveau, 7 belangrijke informatie punten die door geautomatiseerde tools verzorgt kunnen worden, wat ontwikkelaars een hoop tijd bespaart. Hierna, hebben we een focus gelegd op het beoordelen van de test code zelf. Specifiek kijken we naar wanneer en hoe softwareontwikkelaars test code beoordelen, de uitdagingen die hieruit voorkomen, en welke aspecten hiervan ondersteunt kunnen worden door tools. Later, onderzoeken wij de impact van Test-Driven Code Review (TDR) op de effectiviteit van het beoordelen van code, en hoe deze techniek het aantal problemen die gevonden worden in test code kan laten toenemen. Wij bespreken in welke gevallen TDR gebruikt kan worden en waarom niet alle softwareontwikkelaars TDR zien als een toegevoegde waarde.

Acknowledgments

First, I would like to thank all the people involved in the writing and reviewing of this dissertation. Starting from all the committee members, thank you so much for all the feedback you gave me, and thank you for being part of this unique, very important moment of my life. A special thanks to my promotors/advisors Alberto, Magiel, and Arie: thank you for the countless rounds of revisions, feedback, and changes you made on this dissertation. It comes without saying that this thesis would not be the same without your help.

In the next paragraphs, I would like to say few words to *some* of the very important people that helped me during my Ph.D. journey. The emphasis on "some" is because there are probably other 1,000 people that somehow had an impact on me or my carrier: all the researchers I talked to over the last four years, all the discussion during ICSE coffee breaks, or TU Delft coffee breaks, all the visiting researchers, thank you all.

Alberto: Five years ago you sent an email to Prof. Montresor at the University of Trento that literally changed my life. Thank you for allowing me to be part of this amazing group of researchers: I wouldn't be the person I am today if it weren't for you. Thank you so much for all you taught me over the past five years. Thank you for pushing me and motivate me: few people are able to do it as well and effectively as you do. I remember coming in the meetings thinking "I only have six months, I'll never gonna make it!" and coming out thinking "Of course I can make it, what was I thinking?". You always believed in me and pushed me to my best. While writing the last chapter of this dissertation I told you: "Wow, who thought I was able to make it this far?", and you replied "I did." Thank you for that, I will never forget how much you did for me.

Magiel: thank you so much for all the help you gave me in all these years. Moving to another country, my first real job in a company, I need to admit: I was pretty scared. However, you and all the people in SIG welcomed me and throughout the years showed me what "passionate work" actually means. Thank you for being the bridge between industry and academia. Thank you for reminding me of keeping things real, practical, and useful. Thank you for teaching me how to be part of a company. Overall, these last 5 years in SIG were wonderful, and I will bring the memories with me forever. Thank you.

Arie: thank you for giving me the opportunity of being part of the *amazing* SERG group, and thank you for taking good care of it. Being part of this group helped me so much, and I am sure it will help so many other Ph.D. students in the future! It was inspirational, and I will always remember it. Thank you.

Maurício and Laura: you know as well as I do how hard it is to move to another country, leaving your friends and family behind. However, I am happy to say that me and Giada found another family in The Netherlands! Maurício, you and I wrote my first paper together and thinking about it now, I was so fortunate it happened with you! You taught me so much in that first paper, it was an experience I will never forget! You built the foundations of the researcher I am now: it wouldn't be the same without that MSR paper.

Since then, we have shared so many moments! I am sure that in the future, 50 years from now, we will still tell that story of how you, Laura, me, and Giada had that near-death experience in Brazil. Thank you for this and all the other amazing memories.

Anand, Luca, Marco, Vladimir: in the last 4 years we shared the office, taxis, rooms (sometimes even beds :D) but, more importantly, we shared the journey. We were in it together. We drunk when one of us had a paper accepted, we drunk double when one of us had a paper rejected. Thank you for the discussions we had: some of the chapters of this dissertation are the results of them. *Anand*: Thank you for teaching me how to travel like a pro: I never saw someone more efficient than you in using KLM, Hertz, TripAdvisor, etc.! When I was traveling with you, rest assured, I didn't need to move a finger! Just follow Anand, he knows. Thank you, my friend! *Luca* Thank you for cooking one of the best pizzas I've ever had, and for your genuineness. You have been a wonderful companion in this trip. From now on, just remember: when reserving an AirBnb for two people, make sure the beds are separated (or at least a king size bed). *Marco* Thank you for sharing the SIG journey with me. Our italian coffee breaks were really important! *Vladimir*: Thank you for all the discussions on research impact, on SE, on developers, they were really important! Thank you guys!

SERG: a big thank you to all the Ph.D., Post-Doc, and Professors of the SERG group, without you this would not have been possible. Annibale, thank you for all the talks we had. You helped me be more thorough in my studies. Now that our journey separates, I will remember the lessons you taught me. In the words of your favorite character Darth Vader: "When I left you, I was but the learner. Now I am the master." Xavier, Puria, thank you for being very cool persons to hang out with, I hope I will be able to hang out with you in the future!

Giada: a special thank you to my girlfriend Giada, thank you for sharing this journey. We left everything/everyone in Italy and moved to another country, without knowing if we would like it. However, of one thing I was sure: if there were one person on earth I wanted to share this journey with, it was you. Together, we made this possible. Over the past years you showed nothing but unconditional love and support, thank you for that. I am sure the future reserved us other amazing adventures! I love you, and I always will. Thank you.

Dosolina, Enrico, Mirco: posso solo immaginare quanto sia stato difficile vedere un figlio, o un fratello, lasciare tutto e partire per un altro paese. Nonostante questo, mi avete sempre supportato e siete stati al mio fianco. Grazie mille per tutto quello che avete fatto in questi ultimi anni, grazie per tutte le chiamate, il cibo che avete inviato, l'amore che avete dimostrato, anche a distanza. Non lo dimenticherò e ve ne sarò per sempre grato!

Call of Duty team: to Floki, AlexSPARTAN, Cristofernando, Samu, Nuz, and all the other members of our Call of Duty Warzone team, thank you guys for all the nights we spent together playing, joking, camping, cursing (not in this order). From Cristofernando leaving a building Fast-and-Furious-style to Floki leaving all the weapons on the ground and starting slapping everyone in the face: we had a lot of fun! Thank you boyz.

Thanks to all of you. I will forever cherish the memories of my Ph.D. journey.

Davide
Delft, March 2021

1

Introduction

Testing is an essential process in many engineering fields: whether a new television, a car, or a software system, testing can help engineers point out defects and ensure that the product is robust under many usage conditions.

Testing software has the same goal as the testing applied in other engineering fields, namely finding defects or flaws in the product. Testing can be broadly categorized into two areas: manual testing and automated testing. In manual testing, test cases are executed manually by a human without any support from tools or scripts, *i.e.*, a tester manually inspects the product to find the defects, and produce a report afterwords; in automated testing, test cases are executed with the assistance of tools, scripts, and software. Manual testing is the oldest and most rigorous type of software testing. Similar to what it is called "eyeball testing" in other fields, it requires a tester to perform manual test operations on the software: this is very important because no matter how much it improves, automation cannot replace human intuition, inference, and inductive reasoning. For example, a human could change course in the middle of a test run to examine something that had not been previously considered. Furthermore, automated testing cannot test some aspects of the code such as user-friendliness or customer experience. On the other hand, there are some big disadvantages: some type of testing (*e.g.*, load and performance testing) can not be performed manually and, most importantly, manual testing is time consuming and does not scale. That is why companies relies (for the most part) on automated tests, and in this dissertation we mainly focus on this type of testing. In this practice, testers develop test scripts to automatically validate the product. In other words, they do not "manually" test the product, but rely on the pre-scripted tests which will run automatically. Given that the execution of the tests and the computation of their success are automated, automated testing is faster and more reliable than its manual counterpart.

The peculiarity of automated *software* testing is that both the product and the tests are made of the same "material" and are developed in the same way: *the product under test is code (*called *production code), and it is tested using other code (*called *test code).* As one can imagine, software testing involves testing different aspect of the production code: we have non-functional testing (aiming at testing non-functional aspects of the software systems, such as performance, usability, reliability, security), and functional testing which

1

verifies that each function of the software application operates in conformance with the requirements specification. Functional tests are the vast majority of the tests written by developers, and they are the focus of this thesis. In this branch of testing we can find unit testing (tests that exercise a single unit of the software), integration testing (tests that exercise different units and their interaction), system testing, regression testing, smoke testing, BA testing, etc. There are literally hundreds of different types of testing, each one of them measuring different aspects of the production code. If not stated otherwise, in this thesis we do not focus on a specific type, but we take into consideration all the types of tests.

Similarly to engineering in other disciplines, testing of software systems is expected to capture mistakes in the early phases of the development cycle (*e.g.*, before the code is shipped to the clients). Without proper software testing, the mistakes can reach the clients and can be expensive to fix or even dangerous. Examples of critical defects that reached production include NASA, who lost its $125 million Mars Climate Orbiter because of a bug in the conversion from English to metric measurements when exchanging speed and altitude data before the craft was launched [1]. In 2008, Heathrow Terminal 5 was opened to the public in the airport of London, and the baggage handling system collapsed. Over the following 10 days, 42,000 bags failed to travel with their owners, and over 500 flights were cancelled, for a total loss of £4.3bn. Knight Capital Group LLC ("Knight"), one of the biggest American market makers for stocks struggled to stay afloat after a software bug triggered a $440 million loss in just 30 minutes. The firm's shares lost 75% in two days after the faulty software flooded the market with unintended trades. Knight's trading algorithms reportedly started pushing erratic trades through on nearly 150 different stocks.

Despite the expected value of testing, recent studies found that developers perceive and treat test code as less important than production code. For example, van Deursen *et al.* [2] described how the quality of tests was "not as high as the production code [because] test code was not refactored as mercilessly as our production code." A simple explanation of this difference in treatment is that test code, as opposed to production code, is only run by and useful to software engineers, and clients are not aware of how or whether the software is tested; new features, on the other hand, are an easy way to entice more customers and bring more revenue to the company. Joel Spolsky, founder and CEO of StackOverflow, reported how this behavior affected the launch of the very first version of Microsoft Word for Windows [3] and talks about why developers should always prioritize bug fix instead of writing new code.

How can we reduce this difference in treatments and close this gap? This is the focus of this dissertation. We start from the assumption that developers will *always* spend more time in the development of production code, and less time in writing and evaluating tests. Researchers have already investigated the importance of good tests, but the result did not change: production code has still the highest priority. However, *what we can do* is to help developers in spending more intelligently the limited amount of time dedicated to test code. Instead of asking developers to equally split their time between production and test code, we acknowledge that this will not happen and focus on how to help them in spending that limited time dedicated to test code in a better, wiser way. Ultimately, this will lead to better tests that will capture more mistakes in the early phases of the development cycle,

increasing the overall quality of the code. *To achieve this, we put forward the following thesis*:

> **Thesis.** *By devising new techniques and tools, informed by the current practice, to support developers when writing and reviewing test code, we can increase the overall quality of software systems.*

Therefore, we consider two different perspectives:

1. **Writing test code.** In this part of the dissertation, we start investigating how test smells (violations of design principles when writing test code [2]) can impact software code quality, and how we can better present to developers these violations. Later, we study a common practice used when writing test code called Mocking [4], why and how developers use it, and how it is related to software code quality.

2. **Reviewing test code.** In this part of the dissertation, we focus on how to help developers in better reviewing test code. Similar to scientific peer review, Modern Code Review (MCR) [1] is a collaborative process in which reviewers assess whether the code is robust and reliable enough to be accepted and pushed into production. But to what extent is test code considered during code review?
   We first investigate developers information needs when it comes to review code; then, we focus on current best practices when reviewing test code specifically, presenting and evaluating a new practice called *Test Driven Review* (TDR). Finally, we also challenge the current MCR practice, by studying how availability bias can change the outcome of a code review.

1.1 Designing Test Code

Can you imagine spending hours investigating a bug in the production code, only to find out that the error was in the test code instead? The importance of having well-designed code for automated tests was initially put forward by Beck [6]. Beck argued that test cases respecting good design principles are desirable since these test cases are easier to comprehend, maintain, and exploit to diagnose problems in production code. Inspired by these arguments, van Deursen *et al.* [2] coined the term *test smells* and defined the first catalog of eleven poor design choices to write tests, together with refactoring operations aimed at removing them. Such a catalog has been then extended by practitioners, such as Meszaros [7] who defined 18 new test smells.

In the following we present an example of 'Lazy Test', one of the types of test smell originally defined by van Deursen *et al.* [2].

[1] In the context of this dissertation, *Modern* Code Review refers to the more lightweight and informal process of the original Code Inspection process presented by Fagan [5]. In literature, MCR often refers to a Merge-Request development model, where there are 2 versions of the file (before and after a change) and the developer is asked to review the change. More information on MCR with a real example in Section 1.2

1

```
def test_since_filter():
  since_date = datetime(2018, 3, 22, 10, 41, 45)
  repo = RepositoryMining("myrepo", since=since_date)
  l = list(repo.traverse_commits())
  assert len(l) == 3

def test_since_filter1():
  since_date = datetime(2018, 3, 22, 10, 41, 30)
  repo = RepositoryMining("myrepo", since=since_date)
  l = list(repo.traverse_commits())
  assert len(l) == 3

def test_since_filter2():
  since_date = datetime(2018, 3, 22, 10, 42, 3)
  repo = RepositoryMining("myrepo", since=since_date)
  l = list(repo.traverse_commits())
  assert len(l) == 2
```

Example 1.1: A test class suffering of a 'Lazy Test' smell

In the example, we show three test methods that check the same production method "traverse_commits()" with slightly different inputs. In this case, van Deursen *et al.* [2] discussed that tests having this smell are harder to understand, since lot of the code is repeated and a developer would need to read many lines to understand which inputs are covered in the tests and which are not.

In the following, we show an example of refactoring to make the test more robust, by taking advantage of a technique called "parameterized testing":

```
oldest_date = datetime(2018, 3, 22, 10, 41, 30)
middle_date = datetime(2018, 3, 22, 10, 41, 45)
newest_date = datetime(2018, 3, 22, 10, 42, 3)

@pytest.mark.parametrize('since, expected_result', [
  (oldest_date, 3),
  (middle_date, 3),
  (newest_date, 2),
])
def test_since_filter(since, expected_nr_of_commits):
  repo = RepositoryMining("myrepo", since=since)
  commits_found = list(repo.traverse_commits())
  assert len(commits_found) == expected_nr_of_commits
```

Example 1.2: The same test class after the removal of the smell

The new test file now only contains one test method called "test_to_filter()", and runs with three different parameters ("oldest_date", "middle_date", and "newest_date") and should produce three different results (3, 3, and 2). This is easier to understand because all the different inputs are in the same place (from line 6 to line 8) and the developers can immediately see which inputs are covered in the tests.

From the first catalog of smells, Greiler *et al.* [8, 9] showed that test smells affecting test fixtures (*i.e.*, code used to set the world up in a known state and then return it to its original state after the test is complete) frequently occur in a company setting. Motivated

by this prominence, Greiler *et al.* presented TestHound, a tool able to identify fixture-related test smells such as 'General Fixture' or 'Vague Header Setup' [8]. Van Rompaey *et al.* [10] devised a heuristic code metric-based technique to identify two test smell types, *i.e.*,'General Fixture' and 'Eager Test'.

Turning the attention to the empirical studies that had test smells as their object, Bavota *et al.* [11] studied (i) the diffusion of test smells in 18 software projects, and (ii) their effects on software maintenance. They found that 82% of JUnit classes are affected by at least one test smell and that the presence of test smells has a strong negative impact on the comprehensibility of the affected classes [11]. Tufano *et al.* [12] conducted an empirical study aimed at measuring the perceived importance of test smells and their lifespan during the software life cycle. Key results of the investigation indicated that developers usually introduce test smells in the first commit involving the affected test classes, and in almost 80% of the cases the smells are never removed, primarily because of poor awareness of developers. This study strengthened the case for having tools able to automatically detect test smells to raise developers' knowledge about these issues.

Even though previous research investigated various aspects of test smells, we still miss the link between test smells and software quality aspects. What is the impact of test smells on test and production code quality? How can we help developers focus on the most important design smells in their tests, so that they can refactor them? Answering these questions will helps us in gaining further insight of the maintainability aspects of test smells and their impact on the overall code quality of the system.

1.2 Reviewing Code

This section describes the basic components that shape a modern code review, as well as the literature on how code review impacts code quality. As we will see, there is a gap in the literature about reviewing test code specifically.

1.2.1 Background: The code review process

Figure 1.1 depicts a code review (pertaining to the OpenStack project) done with a typical code review tool called Gerrit. Although this is one of the many available review tools, their functionalities are largely the same [13]. In the following we briefly describe each of the components of a review as provided by code review tools.

Code review tools provide an ID and a status (part ① in Figure 1.1) for each code review, which are used to track the code change and know whether it has been *merged* (*i.e.*, put into production) or *abandoned* (*i.e.*, it has been evaluated as not suitable for the project). Code review tools also allow the change author to include a textual description of the code change, to provide reviewers with more information on the rationale and behavior of the change. The second component of a typical code review tool is a view on the technical meta-information on the change under review (part ② in Figure 1.1). This meta-information includes author and committer of the code change, commit ID, parent commit ID, and change ID, which can be used to track the submitted change over the history of the project.

Part ③ of the tool in Figure 1.1 reports more information on who are the reviewers assigned for the inspection of the submitted code change, while part ④ lists the source

1

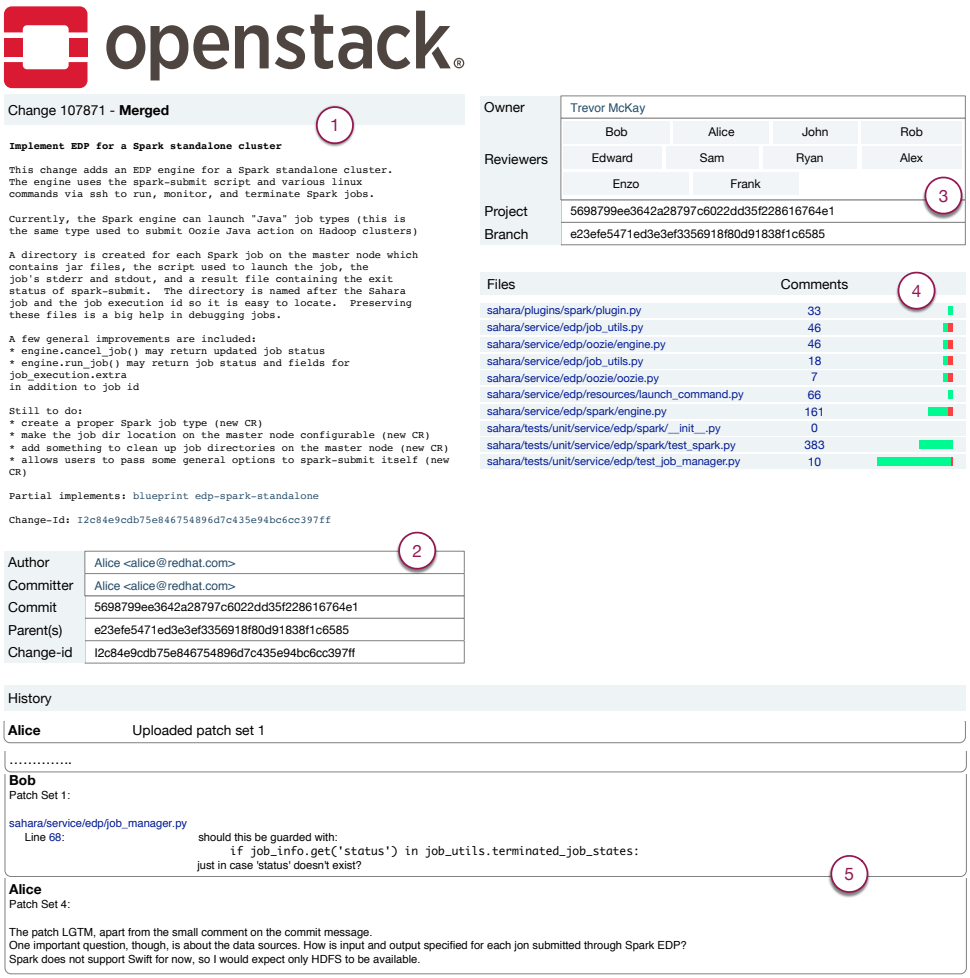


Figure 1.1: Example of code review done with Gerrit.

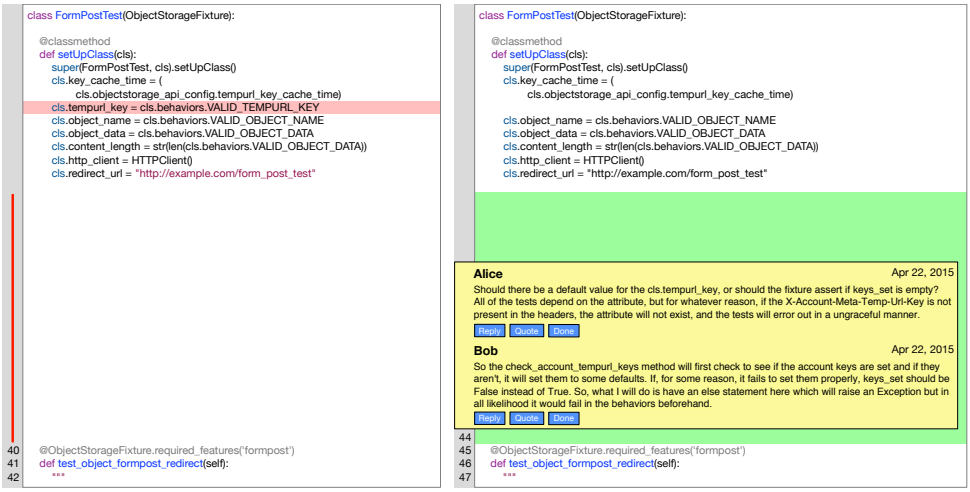


Figure 1.2: Example of code review comments mined from Gerrit.

code files modified in the commit (*i.e.*, the files on which the review will be focused).

Finally, part ⑤ is the core component of a code review tool. It reports the discussion that author and reviewers are having on the submitted code change. In particular, reviewers can ask clarifications or recommend improvements to the author, who can instead reply to the comments and propose alternative solutions. This mechanism is often accompanied by the upload of new versions of the code change (*i.e.*, revised patches or *iterations*), which leads to an iterative process until all the reviewers are satisfied with the change or decide to not include it into production. Figure 1.2 shows a different view that contains both reviews and author comments. In this case, the involved developers discuss a specific line of code, as opposed to Alice from the previous example who commented on the entire code change (Figure 1.1, end of part ⑤).

1.2.2 Empirical Knowledge on Modern Code Review

Over the last decade the research community spent a considerable effort in studying code reviews (*e.g.*, [14–21]). In this section, we describe some of the most important research studies on different areas of MCR. Specifically, we describe how code review participation has an impact on the effectiveness of the review, how code review is linked to better software code quality, and which human aspects can have an impact on the code review performance.

Code Review Participation Extensive work has been done by the software engineering research community in the context of code review participation and how it impacts the code review outcome. Abelein *et al.* [22] investigated the effects of user participation and involvement on system success and explored which methods are available in literature, showing that it can have a significant correlation with system quality. Thongtanunam *et al.* [23] showed that reviewing expertise can reverse the association between authoring expertise and defect-proneness. Rigby *et al.* [24] reported that the level of review participation is the most influential factor in the code review efficiency. Furthermore, several

studies have suggested that patches should be reviewed by at least two developers to maximize the number of defects found during the review, while minimizing the reviewing workload on the development team [25–28].

In a study of code review practices at Google, Sadowski *et al.* [29] found that Google has refined its code review process over several years into an exceptionally lightweight one, which–in part–seems to contradict the aforementioned findings. Although the majority of changes at Google are small (a practice supported by most related work [30]), these changes mostly have one reviewer and have no comments other than the authorization to commit. Ebert *et al.* [31] made the first step in identifying the factors that may confuse reviewers since confusion is likely impacts the efficiency and effectiveness of code review. In particular, they manually analyzed 800 comments of code review of Android projects to identify those where the reviewers expressed confusion. Ebert *et al.* found that humans can reasonably identify confusion in code review comments and proposed the first binary classifier able to perform the same task automatically; they also observed that identifying confusion factors in inline comments is more challenging than general comments.

Code Review and Software Code Quality. McIntosh *et al.* [32] investigated the impact of MCR practices on software quality. Through a case study of three large open source systems, the authors explored the relationship between post-release defects and: (1) code review coverage, *i.e.*, the proportion of changes that have been code reviewed, (2) code review participation, *i.e.*, the degree of reviewer involvement in the code review process, and (3) code reviewer expertise, *i.e.*, the level of domain-specific expertise of the code reviewers. The authors found that all these code review metrics share a significant link with software quality.

Thongtanunam *et al.* [33] compared MCR practices of defective and clean source code file, showing that future-defective files tend to undergo reviews that are less intensely scrutinized, having less team participation, and a faster rate of code examination than files without future defects. Bavota *et al.* [11] also found that the patches with low number of reviewers tend to have a higher chance of inducing new bug fixes.

Human Aspects of Code Review. Past research has provided evidence that human factors determine code review performance to a significant degree and that code review is a collaborative process [21]. In their study, the authors revealed that besides finding defects and ensuring maintainability, motivations for reviewing code are knowledge transfer (*e.g.*, education of junior developers) and improving shared code ownership, which is closely related to team awareness and transparency.

Besides being a collaborative activity, code review is also demanding from a cognitive point of view for the individual reviewer. A large amount of research is focused on improving code review tools and processes based on the assumption that reducing reviewers' cognitive load improves their code review performance [33, 34]. For instance, Baum *et al.* [35] argue that the reviewer and review tool can be regarded as a joint cognitive system, also emphasizing the importance of off-loading cognitive process from the reviewer to the tool. Baum *et al.* [36] conducted experiments to examine the association of working memory capacity and cognitive load with code review performance. They found that working memory capacity is associated with the effectiveness of finding de-localized defects. However, the authors could not find substantial evidence on the influence of change part ordering on mental load or review performance.

Thongtanunam and Hassan [37] investigated the relationship between the evaluation decision of a reviewer (accept or reject) and the visible information about a patch under review (*e.g.*, comments and votes by prior co-reviewers) [37]. With an observational study on tens of thousands of patches from two popular open-source software systems, Thongtanunam and Hassan found that (1) the amount of feedback and co-working frequency between reviewer and patch author are highly associated with the likelihood of the reviewer providing a positive vote and that (2) the proportion of reviewers who provided a vote consistent with prior reviewers is significantly associated with the defect-proneness of a patch (even though other factors are stronger).

While all these studies investigated different aspects of MCR, we still do not know how this practice is used on test code. How are tests reviewed? What are the pitfalls and challenges developers face when reviewing test code? Are there specific issues that reviewers look for in test files? The answers to these questions can lead to improved code review tools and validated practices which in turn may lead to higher code quality overall.

1.3 Research goal and questions

To provide evidence towards our thesis, and support developers in writing and reviewing test code, we seek to answer five research questions.

First, we want to investigate whether and to what extent poor test code design is associated to software code quality. By studying this relation we can expose the test issues associated to the highest maintainability impact, and present developers with feedback they can apply to solve them. To this aim, we analyze the relationship between the presence of smells in test methods and the quality of both test and production code. Subsequently, we create a test smell detector and perform a study using the GitHub-based code quality tool BetterCodeHub (BCH) [2]. Hence, our first research question is:

> **Research Question 1.** What is the impact of poor test design on code quality and how can we present this feedback to developers?

Second, we focus on current practices to help developers write better test code. Current practices as employed by successful projects can help novice developers understand how to handle specific problems when writing and testing software and expert developers understand whether their practices are in line with others. Among the many practices to write tests, we focus on one that is widely adopted by software developers: mocking. This practice eases the process of unit testing by simulating the dependencies of an object: Given the relevance of mocking, technical literature describes how mocks can be implemented in different languages [7, 38–42]. However, how and why practitioners use mocks, what kind of challenges developers face, and how mock objects evolve over time are still unanswered questions. We aim to fill this gap by studying current best practices, and how the usage of mocks is related to to software code quality.

[2] https://bettercodehub.com

1

Research Question 2. How do developers use Mocks in test code and why? Do mocks help in achieving an higher software quality?

After investigating how developers write test code, we move to another well-established practice adopted in both open source and industry projects: Modern Code Review (MCR). More specifically, we want to investigate the current problems faced by developers when reviewing test code, and how to better support this practice. We start by investigating the information that reviewers need do conduct a code review in general. By investigating reviewers' information needs, we better understand the code review process, as to guide future research efforts and envision how tool support can make developers become more effective and efficient reviewers. To investigate reviewers' information needs we analyze the discussions among the reviewers and the author that happen at code review time. We ask:

Research Question 3. What information do developers need to conduct a proper code review?

After studying reviewers' needs, we focus on how developers review test code specifically. As we saw in the previous chapters, research on MCR mainly focused on how developers perform code review on production code, neglecting test code. We aim to increase our understanding of whether and how code review is also used for ensuring the quality of the tests, and what developers think and do when it comes to reviewing test code. With this we can reveal the challenges faced during reviews and uncover how tools and mechanisms can support the review of test code.

Research Question 4. Why and how do developers review test code?

By studying code review of test code, we unveil a practice that we call *Test-Driven Review* (TDR). In this practice, developers start the review by reading the tests firsts, and later move to reviewing the production code. To study the effects of TDR, we devised an online controlled experiment. While we found that TDR can have a positive effect on the code review effectiveness, it was studied in isolation. In real life (*e.g.*, not in a lab setting), other factors could have an impact on the code review outcome. For example, could other reviewers' comments influence the outcome of the review? This could have an impact on TDR and its effectiveness as well. With this final research question we challenge the MCR practice: currently, using a software review tool, the reviewers and the author conduct an asynchronous online discussion to collectively judge whether the proposed code change is of sufficiently high quality. This practice is quite different from, for example, a peer review setting for scientific articles, where the reviewers normally judge (at least initially) the merit of the submitted work independently from each other. Hence, since reviewers' comments are immediately visible as they are written by their authors, could this visibility bias the other reviewers' judgment, thus affecting the effectiveness of the code review? This brings us to the final research question:

Research Question 5. Does availability bias affect the code review outcome?

1.4 Methodology

Empirical Software Engineering is a field of software engineering focused on gathering data, through measurements and experiments, to build theories about processes involved in software engineering. Our dissertation is embedded in the context of empirical software engineering. We use a mixed-method approach to answer our research questions, employing surveys, controlled experiments, interviews, and mining software repositories [43]. In this section we explain the main characteristics of each research method we used:

- **Mining Software Repositories (MSR).** MSR is a field of software engineering research in which researchers extracts data from software repositories. In this thesis, we mainly focus on two software repositories: Version Control Systems and Code Review repositories. More specifically, we use Version Control repositories to extract information about different aspects of the code and how it evolves over time: First, we build a framework to ease this mining process and present it in Chapter 2, then we use it to study test design issues (Chapter 3, 4) and how developers use mocks (Chapter 5). We mine Code Review repositories to get information on the code review practices of a project, such as the discussion among reviewers (Chapter 6), and how they perform code review on test code (Chapter 7).

- **Interviews and Surveys.** While through MSR researchers can understand *what* happens in a software system, understanding *why* that happens is often not possible by only observing the data. To obtain richer data to answer this kind of questions we use interviews: we can learn why something happened by talking to developers, we can ask their opinions on certain behavior, and understand the decision-making process behind a change. Furthermore, even though interviews are an important source of information, the results might not be generalizable: hence, we also employ surveys to challenge and expand our results. In Chapter 5 we interview four and survey 105 developers, in Chapter 6 we interview seven developers, in Chapter 7 we interview 12 developers, and in Chapter 8 we interview nine and survey 103 developers.

- **Controlled Experiment.** In this dissertation, we devise and run two controlled experiments. A controlled experiment is a scientific test done under controlled conditions, meaning that just one (or a few) factors are changed at a time, while all others are kept constant. In this type of experiment, the participants are split into two (or more) groups: one *control group* (serving as a baseline) and one or more *treatment groups*, in which all variables are identical to the control group except for one, which is the factor being tested. In Chapter 7 we study the effect of TDR: in this case, the variable being tested is the adoption of TDR itself. In Chapter 8 we study the effect of existing review comments on the review outcome: in this case, the variable is the presence of a comment when starting the code review.

1

Finally, this dissertation was carried on in an industrial setting. For the entire duration of his Ph.D., the main author of this dissertation was an employer of Software Improvement Group (SIG), a consultancy company on software code quality located in Amsterdam, The Netherlands. SIG was deeply involved in the development of this dissertation and, as we will see in the following chapters, it allowed the authors to interview their developers, extract anonymized data, or perform controlled experiments within the company. This collaboration between TU Delft and SIG enabled the authors to produce research that also has relevance and value for the industrial context.

Table 1.1: Relation between research questions and chapters

	Research Question	Chapters
1.	What is the impact of poor test design on code quality and how can we present this feedback to developers?	3, 4
2.	How do developers use Mocks in test code and why? Do mocks help in achieving an higher software quality?	5
3.	What information do developers need to conduct a proper code review?	6
4.	Why and how do developers review test code?	7, 8
5.	Does availability bias affect the code review outcome?	9

1.5 Research outline

Table 1.1 illustrates the connection between the chapters in this thesis and the research questions. In the following we detail each chapter.

1.5.1 Tool for Mining Software Repositories

In the first part of the thesis we present PyDriller, a tool for mining Git repositories that we developed to ease the process of mining and used in the subsequent chapters.

Chapter 2 presents PyDriller, an open source Python framework for mining software repositories (MSR). MSR is a technique that analyzes the data available in software repositories, such as version control repositories, mailing list archives, and bug tracking systems. Given the enormous amount of data that researchers have at their disposal nowadays, MSR has become an important field of software engineering research, and among the different sources of information researchers can use, version control systems such as Git are among the most used ones. However, extracting data from Git repositories can be difficult: only a few tools that can interact with Git exist and they are complex and hard to use. PyDriller aims at providing developers with simple APIs to extract information from a Git repository such as commits, developers, modifications, diffs, and source code, thus streamlining the main step of most MSR-based studies. Currently, PyDriller is downloaded ≈80K per month; it is used by many researchers and companies such as SIG and Mozilla, it has 13 contributors and ≈350 stars on GitHub.

1.5.2 Test Code Quality and Software Quality

Chapters 3-5 address the first two research questions, where we investigate the impact of poor test design on the overall code quality of the system, as well as a study on an important software testing practice and how it is related to software code quality.

Chapter 3 explores the relationship between test smells and software code quality. This helps us in gaining further insight on the maintainability aspects of test smells and their impact on the overall code quality of the system. We quantitatively investigate the relationship between the presence of smells in test methods and the change- and defect-proneness of both these test methods and the production code they intend to test. We collect test smell information about ten OSS projects and their 221 major releases. Then, we calculate change- and defect-proneness of all the methods of these systems. Finally, we investigate whether a relation between smelly tests and change/defect proneness exists. We find that tests affected by test smells are associated with higher change- as well as defect-proneness, and tests affected by more smells are associated with a slightly higher change-proneness than methods with fewer smells. As for the production code, we find that it is 59% more likely to contain defects when tested by smelly tests.

Chapter 4 investigates new severity thresholds for test smells. Even though we showed the negative impact that test smells can have on the software code quality, test smells are hardly ever removed through refactoring. One possible reason is that current test smells detector lack severity thresholds: it is reasonable to think that not all the smells have the same impact on test quality and some smells should have the priority on the refactoring. In this study we calibrate the detection thresholds such that severity levels can be assigned to a test smell instance, thus allowing developers to focus on the higher severity smells first. We start by analyzing 1,489 open source projects and use these systems as a benchmark to derive the values for three severity thresholds ('Medium', 'High', 'Very high'). Our results show that for four out of nine test smells the new thresholds are substantially different from the ones previously used in research. Then, we modify a test smell detector tool with the newly derived thresholds and we deploy it in BetterCodeHub (BCH) [44], a web-based code quality analysis tool provided by SIG, a company setting in which part of this dissertation was conducted. We ask BCH users to give feedback on the new thresholds, by marking a test smells instance as false positive or give a rating of its impact on code maintainability. Our results show that the new thresholds lead to a lower amount of false positive. Furthermore, we analyze which test smells have the highest priorities in refactoring and the highest impact on maintainability according to the users. Our results show that 'Empty Test', 'Sleepy Test', and 'Mystery Guest' are the smells with the highest refactoring priority according to our users, while 'Empty Test', 'Ignored Test', and 'Conditional Test Logic' are the smells considered to have the highest impact on the maintainability of the code.

Chapter 5 outlines why and how developers make use of mocks in test code, and how this has an impact on quality of test code. When testing a unit (*e.g.*, a production method), the developer can decide to (1) instantiate all the dependencies (*e.g.*, similar

to real setting), or (2) to mock them: in this case, the inefficiency of slow dependencies such as web services or databases are mitigated, and the developer can focus on testing the specific unit. In this chapter we analyze how and why developers decide to mock: To this aim, we perform a two-phase study. First, we analyze 2,178 test dependencies from three OSS projects and one industrial system, investigating which classes are the most mocked. Then, we interview one developer per system to gain further evidence on why some classes were mocked more than others, and which rules they applied when deciding whether to mock or not. Finally, we corroborate our results with a survey with 105 developers. Our study reveals that the usage of mocks is highly dependent on the responsibility and the architectural concern of the class. Developers report to frequently mock dependencies that make testing difficult (e.g., infrastructure-related dependencies) and to not mock classes that encapsulate domain concepts/rules of the system. Among the key challenges, developers report that maintaining the behavior of the mock compatible with the behavior of original class is hard and that mocking increases the coupling between the test and the production code. Their perceptions are confirmed by our data, as we observed that mocks mostly exist since the very first version of the test class, and that mocks tend to remain for its whole lifetime, and that changes in production code often force the test code to also change.

1.5.3 Reviewing Test Code

In Chapters 6-9 we focus on how we can help developers in better reviewing test code. First, we investigate developers' needs when it comes to code reviewing. Then, we focus on code review of test code specifically: we first study when and how developers review test code, and later we investigate how Test-Driven Code Review (TDR) can have an impact on the code review effectiveness. Finally, we study whether availability bias can have an impact on the effectiveness of a code review.

- **Chapter 6** explores developers' information needs when code reviewing. In Modern Code Review, reviewers and authors conduct an online discussion to ensure that the proposed code changes are of sufficiently high quality. In this chapter we focus on investigating the information that reviewers need to conduct a proper code review. We first analyze 900 code review discussions pertaining to three large open-source software projects. Our analysis led to seven high-level information needs that could be addressed through automated tools, saving up time for reviewers. In addition, we conduct four semi-structured interviews with developers from the considered systems and one focus group with developers from SIG, both to challenge our outcome and to discuss developers' perceptions. Among our results, we found that the needs to know (1) whether a proposed alternative solution is valid and (2) whether the understanding of the reviewer about the code under review is correct are the most prominent ones.
- **Chapter 7** investigates why and how developers review test code. Past research focused on how the review is done on production code and its impact on the software code quality. However, our knowledge on whether and how developers do it on test code is still limited. In this chapter, we focus on identifying current practices and

reveal the challenges faced during test code reviews and uncover needs for tools and features that can support the review of test code. We analyze more than 374,071 code reviews related to three open source projects that employ extensive code review and automated testing. We find that test files are almost twice less likely to be discussed during code review when reviewed together with production files, while the number and length of the comments are similar. Then, we interview developers from the four analyzed projects and from a variety of other open source and industry projects to understand how they review test files and the challenges they face. We discovered that one of the main concerns of the developers is to understand whether the test covers all the relevant paths of the production code, while one of the main problems is the limited amount of time developers can spend on reviewing, due to management policies dictating to review production code more carefully than test code.

- **Chapter 8** describes the benefit and challenges of applying TDR. In TDR, a reviewer inspects a patch by examining the changed test code before the changed production code. While briefly described in gray literature, TDR has not been studied rigorously yet and the impact on the code review effectiveness is still unknown. To this aim, we firstly devised a controlled experiment to study whether TDR has an impact on the proportion of defects and maintainability issues found during code review. A total of 92 developers participated online in the experiment, completing 154 code reviews. In the second phase, we investigate problems, advantages, and frequency of adoption of TDR, by means of nine interviews with participants of the experiment and an online survey with 103 respondents. We discovered that with TDR the participants were able to find an higher number of bugs in test code, while this number remain the same in production code. As for the challenges, most developers seem to be reluctant to devote much attention to tests, as they deem production code more important; moreover applying TDR is problematic due to widespread poor test quality (reducing TDR's applicability) and lack of tool support for this practice.

- **Chapter 9** analyzes the effect of availability bias on code review. In this chapter we investigate other factors that might have an impact on TDR and its adoption: more specifically, we study whether other reviewers' comments can influence the outcome of the review. Using current software review tool, the reviewers and the author of a code change conduct an *asynchronous* online discussion to collectively judge whether the proposed code change is of sufficiently high quality. In these code review tools, reviewers' comments are immediately visible as they are written by their authors, and their visibility might bias the other reviewers' judgment. For example, an existing comment may prime new reviewers on a specific type of bug due to the availability bias. This bias is the tendency to be influenced by information that can be easily retrieved from memory (*i.e.*, easy to recall). We conduct a controlled experiment to test the current code review setup and reviewers' proneness to availability bias. More specifically, we examine whether priming a reviewer on a bug type (achieved by showing an existing review comment) biases the outcome of code review. After testing our results for robustness, we could find no evidence indicating that the outcome of the review is biased in the presence of an existing

review comment priming them on a bug type. Only for one bug type, though, we have strong evidence that the behavior of the reviewers changed: When the previous review comment was about a type of bug that is normally not considered during developers' coding/review practices (*i.e.*, checking for 'NullPointerException' on a method's parameters), the reviewers were more likely to find the same type of bug with a strong effect.

1.6 Origin Of Chapters

All chapters of this thesis have been published in peer-reviewed journal and conferences. As a result, each chapter is self-contained with its background, related work, and implications section. Unless otherwise specified, the first author is the main contributor to each work. In the following, the origin of each chapter is explained:

- **Chapter 2** was published in the paper *"PyDriller: Python Framework for Mining Software Repositories"* by *Davide Spadini*, Mauricio Aniche, and Alberto Bacchelli at the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2018 Demonstrations Track.

- **Chapter 3** was published in the paper *"On The Relation of Test Smells to Software Code Quality"* by *Davide Spadini*, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli at the International Conference on Software Maintenance and Evolution (ICSME) 2018.

- **Chapter 4** was published in the paper *"Investigating Severity Thresholds for Test Smells"* by *Davide Spadini*, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli at International Conference on Mining Software Repositories (MSR) 2020.

- **Chapter 5** was published in the paper *"To Mock or Not To Mock? An Empirical Study on Mocking Practices"* by *Davide Spadini*, Mauricio Aniche, Magiel Bruntink, and Alberto Bacchelli at International Conference on Mining Software Repositories (MSR) 2017. This chapter also contains content from the extension of this paper titled *"Mock objects for testing java systems: Why and how developers use them, and how they evolve"* by *Davide Spadini*, Mauricio Aniche, Magiel Bruntink, and Alberto Bacchelli published in Empirical Software Engineering (EMSE) 2019.

- **Chapter 6** was published in the paper *"Information Needs in Contemporary Code Review"* by Luca Pascarella, *Davide Spadini*, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli at the ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW) 2018. For this work, the first two authors contributed equally.

- **Chapter 7** was published in the paper *"When Testing Meets Code Review: Why and How Developers Review Tests"* by *Davide Spadini*, Mauricio Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli at the International Conference on Software Engineering (ICSE) 2018.

- **Chapter 8** was published in the paper *"Test-Driven Code Review: An Empirical Study"* by *Davide Spadini*, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli at the International Conference on Software Engineering (ICSE) 2019.
- **Chapter 9** was published in the paper *"Primers or Reminders? The Effects of Existing Review Comments on Code Review"* by *Davide Spadini*, Gul Calikli, and Alberto Bacchelli at the International Conference on Software Engineering (ICSE) 2020.

1.7 Open Science

The tools used in the various chapters of this thesis have been made publicly available, together with the instructions on how to execute them. We also provide all the data collected in the chapters, unless it comes from industrial environments where confidentiality must be preserved. An overview of the datasets and where to find them can be found in Table 1.2.

Table 1.2: Relation between research questions and chapters

Dataset	Chapter	Host
PyDriller	2	Zenodo [45]
Test Smells to Software Code Quality	3	Zenodo [46]
Investigating Severity Thresholds for Test Smells	4	Zenodo [47]
To Mock or Not to Mock?	5	Zenodo [48]
Information Needs In Contemporary Code Review	6	Zenodo [49]
When Testing Meets Code Review	7	Zenodo [50]
Test-Driven Code Review	8	Zenodo [51]
Primers or Reminders? 🏆	9	Zenodo [52]

[🏆] ACM SIGSOFT Distinguished Artifact Awards

2

PyDriller: Python Framework for Mining Software Repositories

Software repositories contain historical and valuable information about the overall development of software systems. Mining software repositories (MSR) is nowadays considered one of the most interesting growing fields within software engineering. MSR focuses on extracting and analyzing data available in software repositories to uncover interesting, useful, and actionable information about the system. Even though MSR plays an important role in software engineering research, few tools have been created and made public to support developers in extracting information from Git repository. In this chapter, we present PyDriller*, a Python Framework that eases the process of mining Git. We compare our tool against the state-of-the-art Python Framework GitPython, demonstrating that* PyDriller *can achieve the same results with, on average, 50% less LOC and significantly lower complexity.*

URL: https://github.com/ishepard/pydriller,

Preprint: https://doi.org/10.5281/zenodo.1327411

Data and materials: https://doi.org/10.5281/zenodo.1327363,

This chapter has been published as *Davide Spadini, Mauricio Aniche, Alberto Bacchelli. PyDriller: Python Framework for Mining Software Repositories. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, pages 908–911, New York, New York, USA, 2018. ACM Press.* [53]

2.1 Introduction

Mining software repository (MSR) techniques allow researchers to analyze the information generated throughout the software development process, such as source code, version control systems metadata, and issue reports [54–56]. With such analysis, researches can empirically investigate, understand, and uncover useful and actionable insights for software engineering, such as understanding the impact of code smells [57–59], exploring how developers are doing code reviews [19, 21, 28, 60] and which testing practices they follow [20], predicting classes that are more prone to change/defects [61–64], and identifying the core developers of a software team to transfer knowledge [65].

Among the different sources of information researchers can use, version control systems, such as Git, are among the most used ones. Indeed, version control systems provide researchers with precise information about the source code, its evolution, the developers of the software, and the commit messages (which explain the reasons for changing).

Nevertheless, extracting information from Git repositories is not trivial. Indeed, many frameworks can be used to interact with Git (depending on the preferred programming language), such as GitPython [1] for Python, or JGit for Java [2]. However, these tools are often difficult to use. One of the main reasons for such difficulty is that they encapsulate all the features from Git, hence, developers are forced to write long and complex implementations to extract even simple data from a Git repository.

In this chapter, we present PyDriller, a Python framework that helps developers to mine software repositories. PyDriller provides developers with simple APIs to extract information from a Git repository, such as commits, developers, modifications, diffs, and source code. Moreover, as PyDriller is a framework, developers can further manipulate the extracted data and quickly export the results to their preferred formats (*e.g.*, CSV files and databases).

To evaluate the usefulness of our tool, we compare it with the state-of-the-art Python framework GitPython, in terms of implementation complexity, performance, and memory consumption. Our results show that PyDriller requires significantly fewer lines of code to perform the same task when compared to GitPython, with only a small drop in performance. Also, we asked six developers to perform tasks with both tools and found that all developers spend less time in learning and implementing tasks in PyDriller.

2.2 PyDriller

PyDriller is a wrapper around GitPython that eases the extraction of information from Git repositories. The most significant difference between the two tools is that GitPython offers many features (almost all the features of Git), while PyDriller offers only features that are important when performing MSR tasks, thus hiding the underlying complexity to the end user. In this section, we explain the design of PyDriller, as well as its main APIs.

2.2.1 Domain Object

Commit. It contains all the information regarding the commit: the hash, the committer (name and email), the author (name and email), the message, the authored and

[1] https://github.com/gitpython-developers/GitPython
[2] https://www.eclipse.org/jgit/

committed dates, a list of its parents' hashes (a merge commit has two parents), and the list of modified files (see 'Modification' object below). Since loading the entire object is expensive and time consuming (*e.g.*, PyDriller needs to retrieve and parse the diff of the commit), objects are *lazy loaded*, i.e., are only computed when needed.

Modification. This object carries information regarding a file changed in a commit. A modification object has the following fields:

Old path: old path of the file (can be *None* if the file is added).

New path: new path of the file (can be *None* if the file is deleted).

Change type: type: 'Added', 'Deleted', 'Modified', or 'Renamed'.

Diff: diff of the file as Git presents it (starting with *@@ xx,xx @@*).

Source code: source of the file (can be *None* if the file is deleted).

Added: number of lines added.

Removed: number of lines removed.

Filename: The name of the file.

2.2.2 Architecture

RepositoryMining. This class is in charge of running the MSR study. The only required parameter of this class is the path to the Git repository to analyze. Based on the Git path, the framework will return the list of commits in the repository.

Since MSR studies are highly customizable, to allow a researcher to customize the study, we expose a set of APIs , making it possible to set the dates in which PyDriller should start to analyze, as well as filtering only specific commits. The complete list of filters is the following:

Select starting point: *since* (after this date), *from commit* (after this commit hash), and *from tag* (after this commit tag)

Select ending point: *to* (up to this date), *to commit* (up to this commit hash), and *to tag* (up to this commit tag)

Select by commits: *single* (single hash of the commit), *only in branches* (only consider certain branches), *only in main branch* (only commits that belong to the main branch), *only no merge* (only commits that are not merge commits), and *only modifications with file types* (only commits in which at least one modification was done in that file type, *e.g.*, by specifying '.java', only commits with at least one Java file was modified are visited.)

In the following, we present some examples of how metrics can be customized and adapted to various MSR studies:

```
# Analyze single commit
RepositoryMining('path/to/the/repo',
    single='6411e3096dd2070438a17b225f4447')

# Since 8/10/2016
dt1 = datetime(2016, 10, 8)
```

```
RepositoryMining('path/to/the/repo', since=dt1)

# Between 2 dates
dt1 = datetime(2016, 10, 8, 17, 0, 0)
dt2 = datetime(2016, 10, 8, 17, 59, 0)
RepositoryMining('path/to/the/repo', since=dt1, to=dt2)

# Between tags
first_tag = 'tag1'
last_tag = 'tag2'
RepositoryMining('path/to/the/repo', from_tag=first_tag, to_tag=last_tag)

# Only commits in main branch
RepositoryMining('path/to/the/repo', only_in_main_branch=True)

# Only commits in main branch and no merges
RepositoryMining('path/to/the/repo', only_in_main_branch=True, only_no_merge=True)

# Only commits that modified a java file
RepositoryMining('path/to/the/repo', only_modifications_with_file_types=['.java'])
```

After the user configured the *RepositoryMining* class, thus specifying which commits to analyze, the user has only to call the **traverse_commit()** function that will return the desired list of commits. Internally, PyDriller obtains the list of all the commits, filters out the unnecessary ones, converts the commits in a domain object, and returns the list of resulting commits. This approach has the advantage that all the complexity is hidden from users.

Furthermore, if users need more than just visiting commits, we created a wrapper for the most common utilities of Git, for example checkout, reset, log, show a single commit. We also built APIs to help researchers in MSR studies, including:

Parse diff : The diff presented by Git is difficult to parse. With this API, given a diff, it returns a dictionary with the added and deleted lines. For both groups, the function returns a tuple, corresponding to 1) line number in the file and 2) actual line.

Get commits that last modified lines: This function applies SZZ [61]. Given a 'Commit' object as parameter, it returns the set of commits that changed last the lines modified in the files included in the commit. The algorithm works as follow (for every file in the commit): 1) obtain the diff, 2) obtain the list of deleted lines, and 3) blame the file and obtain the commits were those lines were changed last.

To facilitate the data analysis, PyDriller gracefully handles GitPython exceptions. For example, when retrieving the source code of non-UTF-8 files (*e.g.*, bytecodes), GitPython raises an exception, while PyDriller returns an empty string. Hence, PyDriller reduces the burden of handling several exceptions that a developer would have to do otherwise.

2.3 Evaluation

To evaluate our tool, we compare PyDriller against the state-of-the-art Git framework for Python–GitPython. We select five different common MSR tasks that we encountered

in our experience as researchers in the MSR field, and implement them using both frameworks. The tasks follow:

Task 1: Calculating complexity of the added lines for every commit. For the sake of simplicity, we define complexity as the number of *if* statements in the diff.

Task 2: Detecting bug inducing commits. We use SZZ [61] to retrieve the commits where the bug was introduced, as normally done in previous literature [20, 57, 59].

Task 3: Obtaining the list of commits that only modified Java files.

Task 4: Lines of code per source file over time.

Task 5: Day of the week developers fixed more bugs between two releases.

We run the five tasks (implemented in both PyDriller and GitPython) on 50 OSS projects, 25 belonging to the Eclipse Foundation and 25 to Apache. We selected the projects using GHTorrent [66], taking the 25 most starred projects of the two organizations. For the sake of simplicity, we only report the results of one project, Apache Hadoop. The results of the other 49 projects can be found in our on-line appendix [45].

We compare the tools under different metrics: lines of code (LOC) and complexity (McCabe complexity [67]) of both implementations, as well as their memory consumption, and execution time. Table 2.1 shows the results. For all the exercises, both in PyDriller and GitPython, the number of lines that are not a core functionality (for example the constructor) is three. We keep this number as it is always the same for all the exercises and for both tools.

Regarding execution time, PyDriller is generally slower than GitPython. This decrease in speed is expected, given that PyDriller is a wrapper built on top of the python framework. However, the difference is small: In the most expensive tasks (Ex1 and Ex4), in which the tools have to analyze the diff or source code of every file in 20,000 commits, PyDriller is only 1:34 minutes slower in the first case. In the other task, PyDriller is 44 seconds faster than GitPython. Nevertheless, both tools take less than 16 minutes to analyze the entire history of Apache Hadoop (avg. 22 commits per second). As for memory consumption, the tools behave similarly: In some cases, the used memory is less than 50MB. In the most memory consuming task (number 1), the used memory was 169MB.

The large difference between both tools is in terms of LOC and complexity of the implementation. For the former, we see that using PyDriller results (on average) in writing 50% less lines of code than using GitPython. The biggest difference is in the task 2, where the tool had to retrieve the bug inducing commits using the SZZ algorithm: This problem was solved in 19 LOC using PyDriller, while 66 LOC with GitPython (70% difference).

We also observe that the complexity of the code written for PyDriller is significantly lower than for GitPython. Table 2.1 shows that, on average, the code for PyDriller is 60% less complex. This is especially the case in tasks that have to deal with retrieving the diff or source code of the modified files; indeed, obtaining this information in PyDriller is just 1 API call, while GitPython requires many lines of code and exceptions handling.

Table 2.1: Comparison between Pydriller and GitPython.

		Pydriller	GitPython	Total
Ex1	Time	00:14:59	00:13:25	+00:01:34
	Max Memory (MB)	169	148	+21
	LOC	21	54	-61%
	Complexity	7	15	-53%
Ex2	Time	00:01:14	00:01:00	+00:00:14
	Max Memory (MB)	–	–	–
	LOC	19	66	-71%
	Complexity	5	6	Similar
Ex3	Time	00:01:24	00:01:14	+00:00:10
	Max Memory (MB)	94	39	+55
	LOC	10	18	-44%
	Complexity	2	6	-67%
Ex4	Time	00:15:03	00:15:47	-00:00:44
	Max Memory (MB)	162	132	+30
	LOC	17	42	-60%
	Complexity	6	15	-60%
Ex5	Time	00:00:02	00:00:03	Similar
	Max Memory (MB)	–	–	–
	LOC	12	19	-37%
	Complexity	3	4	Similar

2.4 Evaluation with developers

To further evaluate our tool, we invited six developers to perform the same two tasks using both Pydriller and GitPython, and to note the time they took to solve the problems, as well as their personal opinions on both tools. All developers had experience in developing with Python and on performing MSR studies, but they had never used PyDriller nor GitPython before.

We asked the participants to solve tasks 3 and 4. We chose these tasks because they are simpler than the first two (to keep the experiment short) and do not require participants to have notions on how to identify bug fixing commits (Ex5). The setting of the experiment is the following:

- Participants should implement both tasks, first with Pydriller, then with GitPython. Since understanding how to solve the tasks does require some additional time, we asked the participants to start with Pydriller. This choice clearly penalizes our tool, as participants will have a better intuition about the tasks when doing the task in GitPython. However, we believe that Pydriller is simpler to use, and that the difference between the two tools will still be significant.

- Participants should take notes about the time it takes them to implement the tasks. We ask participants to also include the time spent reading the documentation of the two tools, since understanding how to use the tool is part of the experiment.

- After having implemented both tasks, we ask to the participants to elaborate on the different advantages and disadvantages between both tools.

Table 2.2: Time spent by the participants of the experiment in solving tasks 3 and 4 together.

Participant	Time (minutes) with PyDriller	Time (minutes) with GitPython	Total
P1	45	80	-44%
P2	23	45	-49%
P3	13	20	-35%
P4	19	26	-27%
P5	44	46	–
P6	17	30	-43%

The result of the experiment is shown in Table 2.2. Five out of six participants spent significantly less time to solve the problems (27% less in the worst case, 49% less in the best case). P5, instead, solved both problems in the same amount of time: the participant did not know how to solve the second task and, since he started with PyDriller, this translated in more time in the first part. When he understood how to solve it, he moved to GitPython already knowing the solution.

All participants agreed that PyDriller was easier to use than GitPython [P_{1-6}]. P_6 said: *"I thought PyDriller was a lot more intuitive than using GitPython. GitPython works exactly like Git, so it isn't very well suited when trying to gain insights about the repository."*

Similarly, P_1 affirmed that, using PyDriller, he was able to achieve the same result with simpler and shorter code, and that he will continue to use PyDriller in his next MSR studies. P_2 added that GitPython is useful when one has to simulate Git commands in Python, but it can be overcomplicated when the goal is to perform MSR studies, for which PyDriller is more appropriate, because it hides this complexity from its users.

2.5 Related Tools

In this section we compare PyDriller against two of the most recent and used MSR tools.
Boa [68]: Boa is a domain-specific language and infrastructure that eases MSR. The main difference between PyDriller and Boa is that, while the former can be run on every project, Boa can only be used on their snapshots of GitHub or SourceForge, which currently are 3 and 6 years old. Furthermore, PyDriller is written in Python. Hence, it has all the flexibility of a the programming language and can be used together with other frameworks. Boa, on the other hand, has its own DSL, and can not be used with other (external) libraries. Furthermore, Boa currently includes only the history and source code of Java projects, while PyDriller can be used to analyze repositories of any programming language.
GHTorrent [66]: GHTorrent is a scalable, queryable, offline mirror of data present on GitHub. The main difference between PyDriller and GHTorrent is that, while the former retrieves all the information regarding a commit (*e.g.*, what files changed, diffs, and source code), the latter focuses on GitHub's social data, such as pull requests, issues, and users. However, GHTorrent does not offer the possibility of navigating through the commits or analyzing the project's source code over time (which is a feature of PyDriller).

2.6 Conclusion

In this chapter, we presented Pydriller, a Python framework that helps developers on mining software repositories. We showed that with Pydriller, developers can easily extract information from any Git repository, such as commits, developers, modifications, diffs, and source codes, since Pydriller releases simple APIs to help researchers and practitioners performing MSR.

We evaluated Pydriller on 5 exercises, comparing it against GitPython. The evaluation showed that using Pydriller results in writing (on average) half the code, and 60% less complex. Furthermore, we asked 6 developers to solve two exercises using our tool, and they all agreed that Pydriller helped them in solving the problems in less time with less code.

The first version of PyDriller has been released on 9th April 2018, and since then it has been downloaded approximatively 1,000 times per month (as computed through "Pypinfo" [3] and Google BigQuery). We plan to keep improving Pydriller's performance as well as to perform more user studies with the goal of understanding even better what MSR researchers require in their studies.

[3] https://github.com/ofek/pypinfo

3

On The Relation of Test Smells to Software Code Quality

Test smells are sub-optimal design choices in the implementation of test code. As reported by recent studies, their presence might not only negatively affect the comprehension of test suites but can also lead to test cases being less effective in finding bugs in production code. Although significant steps toward understanding test smells, there is still a notable absence of studies assessing their association with software quality.

In this chapter, we investigate the relationship between the presence of test smells and the change- and defect-proneness of test code, as well as the defect-proneness of the tested production code. To this aim, we collect data on 221 releases of ten software systems and we analyze more than a million test cases to investigate the association of six test smells and their co-occurrence with software quality. Key results of our study include:(i) tests with smells are more change- and defect-prone, (ii) 'Indirect Testing', 'Eager Test', and 'Assertion Roulette' are the most significant smells for change-proneness and, (iii) production code is more defect-prone when tested by smelly tests.

Preprint: https://doi.org/10.5281/zenodo.1689875,

Data and materials: https://doi.org/10.5281/zenodo.4075336.

This chapter has been published as *Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the Relation of Test Smells to Software Code Quality. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 1–12. IEEE.*[69]

3.1 Introduction

Automated testing (hereafter referred to as just *testing*) has become an essential process for improving the quality of software systems [70, 71]. In fact, testing can help to point out defects and to ensure that production code is robust under many usage conditions [70, 72]. Writing tests, however, is as challenging as writing production code and developers should maintain test code with the same care they use for production code [73].

Nevertheless, recent studies found that developers perceive and treat production code as more important than test code, thus generating quality problems in the tests [74–77]. This finding is in line with the experience reported by van Deursen *et al.* [2], who described how the quality of test code was "not as high as the production code [because] test code was not refactored as mercilessly as our production code" [2]. In the same work, van Deursen *et al.* introduced the concept of *test smells*, inspired by Fowler *et al.*'s *code smells* [78]. These smells were recurrent problems that van Deursen *et al.* found when refactoring their troublesome tests [79].

Since its inception, the concept of test smells has gained significant traction both among practitioners [7] and the software engineering research community [2, 9–11]. Bavota *et al.* presented the earliest and most significant results advancing our empirical knowledge on the effects of test smells [11]. The researchers conducted the first controlled laboratory experiment to establish the impact of test smells on program comprehension during maintenance activities and found evidence of a negative impact of test smells on both comprehensibility and maintainability of test code [11].

Although the study by Bavota *et al.* [11] made a first, necessary step toward the understanding of maintainability aspects of test smells, our empirical knowledge on whether and how test smells are associated with software quality aspects is still limited. Indeed, van Deursen *et al.* [2] based their definition of test smells on their anecdotal experience, without extensive evidence on whether and how such smells are negatively associated with the overall system quality.

To fill this gap, in this chapter we quantitatively investigate the relationship between the presence of smells in test methods and the change- and defect-proneness of both these test methods and the production code they intend to test. Similar to several previous studies on software quality [80, 81], we employ the proxy metrics change-proneness (*i.e.*, number of times a method changes between two releases) and defect-proneness (*i.e.*, number of defects the method had between two releases). We conduct an extensive observational study [82], collecting data from 221 releases of ten open source software systems, analyze more than a million test cases, and investigate the association between six test smell types and the aforementioned proxy metrics.

Based on the experience and reasoning reported by van Deursen *et al.* [2], we expect to find tests affected by smells to be associated with more changes and defects, *i.e.*, higher maintenance efforts and lower software quality. Furthermore, since test smells indicate poor design choices [2] and previous studies showed that better test code quality leads to better productivity when writing production code [83], we expect to find production code tested by smelly tests to be associated with more defects.

Our results meet these expectations: Tests with smells are more change- and defect-prone than tests without smells and production code is more defect-prone when tested by smelly tests. Among the studied test smells, 'Indirect testing', 'Eager Test' and 'Assertion

Roulette' are those associated with highest change-proneness; moreover, the first two are also related to a higher defect-proneness of the exercised production code. Overall, our results provide empirical evidence that detecting test smells is important to signal underlying software issues as well as studying the interplay between test design quality and effectiveness on detecting defects is of paramount importance for the research community.

3.2 Research Methodology

The *goal* of our study is to increase our empirical knowledge on whether and how test methods affected by smells are associated with higher change- and defect- proneness of the test code itself, as well as to assess whether and to what extent test methods affected by test smells are associated with the defect-proneness of the production code they test. The *perspective* is that of both researchers and practitioners who are interested in understanding the possible adverse effects of test smells on test and production code. We structured our study around the two overarching research questions that we describe in the following.

The first research question investigates the relationship between the presence of test smells in test code and its change/defect proneness:

> **RQ1.** *Are test smells associated with change/defect proneness of test code?*

We, thus, structure **RQ1** in three sub-research questions. First, we aim at providing a broad overview of the relationship of test smells and their co-occurrence with change- and defect-proneness of test code:

RQ1.1: *To what extent are test smells associated with the change- and defect- proneness of test code?*

RQ1.2: *Is the co-occurrence of test smells associated with the change- and defect-proneness of test code?*

Then, we aim at verifying whether some particular test smells have a stronger association with change- and defect- proneness of test code:

RQ1.3: *Are certain test smell types more associated with the change- and defect-proneness of test code?*

Considering that defect-proneness as been widely used in previous literature as a proxy metric for software quality (*e.g.*, [63, 80, 84, 85]), in the second research question, we aim at making a complementary analysis into the association of test smells with the defect-proneness of the exercised production code. In fact, if the production code exercised by tests with test smells is more defect-prone this would be an even stronger signal on the relevance of test smells. This goal leads to our second research question:

RQ2. *Is the production code tested by tests affected by test smells more defect-prone?*

The expectation is that test code affected by test smells might be less effective in detecting defects [83], thus being associated with more defect-prone production code. We structured **RQ2** in three sub-research questions:

RQ2.1: *Are test smells associated with the defect-proneness of the tested production code?*

RQ2.2: *Is the co-occurrence of test smell associated with the defect-proneness of the tested production code?*

RQ2.3: *Are certain test smell types more associated with the defect-proneness of production code?*

Similarly to **RQ1**, we aim at providing an overview of the role of test smells in the defect-proneness of production code, by investigating single test smells and their co-occurrence.

Table 3.1: Subject systems' details

System	#Releases	#Classes (Min-Max)	#Methods (Min-Max)	#KLOC (Min-Max)
Apache Ant	10	9-282	74-2,541	1-25
Apache Cassandra	25	61-437	237-4,804	2-59
Apache Hadoop	35	470-1,895	3,400-19,445	71-344
Apache Wicket	44	102-585	587-3,351	8-46
Eclipse JDT	17	11-56	68-4,068	1-49
ElasticSearch	36	25-698	324-6,755	5-118
Hibernate	8	823-1,508	5,461-9,027	92-144
Sonarqube	36	492-2,072	2,256-18,028	18-134
Spring Framework	7	980-1,662	10,576-18,049	136-212
VRaptor4	3	122-125	1,046-1,102	8-9
Total	221	9-2,072	68-19,445	1-344

3.2.1 Subjects of the Study

In our study, we have to select two types of subjects: software systems and test smells.

Software systems. We consider ten OSS projects and their 221 major releases as subject systems for our study. Specifically, Table 3.1 reports the characteristics of the analyzed systems concerning (i) the number of the considered releases and (ii) size, in terms of the number of classes, methods, and KLOCs. Two main factors drive the selection: firstly, since we have to run static analysis tools to detect test smells and compute maintainability metrics, we focus on projects whose source code is publicly available (*i.e.*, OSS); secondly, we analyze systems having different sizes and scopes. After filtering on these criteria, we randomly select ten OSS projects from the list available on GITHUB[1] having different size, scope, and with a number of JUnit test cases higher than 1,000 in all the releases.

[1]https://github.com

For each system, we only consider their major releases. In fact, (i) detecting test smells at commit-level is prohibitively expensive in terms of computational time and (ii) minor releases are too close to each other (in some cases there is more than one minor release per week), so very few changes are made in the source and test code. We mine these major releases directly from the systems' GITHUB repositories.

Test smells. As subject test smells for our study, we consider those described in Table 3.2. While other test smell types have been defined in literature [2, 7], we select the smells in Table 3.2 because: (1) Identifying test smells in 221 project releases through manual detection is prohibitively expensive, thus a reliable and accurate automatic detection mechanism must be available; (2) the selected test smells have the greatest diffusion in industrial and OSS projects [11]; and (3) the selected ones compose a diverse catalog of test smells, which are related to different characteristics of test code.

3.2.2 Data Extraction

To answer **RQ1**, we extract data about (i) the test smells affecting the test methods in each system release and (ii) the change/defect proneness of these test cases. To answer **RQ2**, we extract data about the defect proneness of the production code exercised by the test code. The obtained data and the *R* script used to analyze the results are both available in our online appendix [46].

Detecting test smells. We adopt the test smell detector by Bavota *et al.* [11] (widely adopted in previous research [11, 12, 86]), which is able to reliably identify the six smells considered in our study with a precision close to 88% and a recall of 100%, by relying on code metrics-based rules.

Defining the change-proneness of test code. To compute change- and defect-proneness of test code, we mine the change history information of the subject systems using REPODRILLER [87], a Java framework that allows the extraction of information such as commits, modifications, diffs, and source code. Explicitly, for each test method T_i of a specific release r_j we compute its change-proneness as follows:

$$change\_proneness(T_i, r_j) = \#commits(T_i)_{r_{j-1} \rightarrow r_j}$$

where $\#commits(T_i)_{r_{j-1} \rightarrow r_j}$ represents the number of changes performed by developers on the test method T_i between the releases r_{j-1} and r_j. Given the granularity of our analyses (*i.e.*, release-level), we only compute the change-proneness of test methods that were actually present in a release r_j; if a new method was added and removed between r_{j-1} and r_j, it does not appear in our result set. To identify which test method changed within a commit, we implement the following algorithm:

1. We first identify all test classes modified in the commit. In line with past literature [12, 56], we consider a class to be a test when its name ends with 'Test' or 'Tests'.

2. For each test class, we obtain the source code of the class in both the present commit and the previous one.

3. We parse the source code of the test class to identify the test methods contained in the current and in the previous commit. Then, we compare the source code of

each test method from the current commit against all the test methods of the prior version:

(a) if we find the same method, it means that it is not changed (*i.e.*, both signature and content of the method in r_j are the same as r_{j-1});

(b) if we find a different method, it means that it is changed (*i.e.*, the signature of the method is the same, but the source code in r_j is not equal to r_{j-1});

(c) if we do not find the method (*i.e.*, the signature of the method does not exist in the previous version of the file), it means that it has been added or renamed. To capture the latter, we adopt a technique similar to the one proposed by Biegel *et al.* [88], based on the use of textual analysis to detect rename refactoring operations. Specifically, if the cosine similarity [89] between the current method and that of the methods in the previous version is higher than 95%, then we consider a method as renamed (hence, it inherited all the information of the old test case).

Defining the defect-proneness of test code. To compute the defect-proneness of each test case, we follow a similar procedure to the one for change-proneness, with the exception that to calculate the *buggy* commits we relied on SZZ [61]. In particular, we first determine whether a commit fixed a defect employing the technique proposed by Fischer *et al.* [90], which is based on the analysis of commit messages. If a commit message matches an issue ID present in the issue tracker or it contains keywords such as *'bug'*, *'fix'*, or *'defect'*, we consider it as a bug fixing activity. This approach has been extensively used in the past to determine bug fixing changes [91, 92] and it has an accuracy close to 80% [57, 90], thus we deem it as being accurate enough for our study. Once we have detected all the bug fixing commits involving a test method, we employ SZZ to obtain the commits where the bug was introduced.

To estimate the moment when a bug was likely introduced, the SZZ algorithm relies on the annotation/blame feature of versioning systems [61]. In short, given a bug-fix activity identified by the bug ID k, the approach works as follows:

- For each file f_i, $i = 1 \dots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k) and fixed in its revision $\textit{rel-fix}_{i,k}$, we extracted the file revision just *before* the bug fixing ($\textit{rel-fix}_{i,k} - 1$).

- Starting from the revision $\textit{rel-fix}_{i,k} - 1$, for each source line in f_i changed to fix the bug k, we identified the production method M_j to which the changed line changed belongs. Furthermore, the `blame` feature of `Git` is used to identify the revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser [93]. This produces, for each production method M_j, a set of $n_{i,k}$ bug-inducing revisions $\textit{rel-bug}_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

With the list of bug inducing commits involving every test method, we compute its defect-proneness in a release r_j as the number of bug inducing activities involving the method in the period between the releases r_{j-1} and r_j .

Defining the defect-proneness of production code. For each test method in the considered projects, we first need to retrieve what is the production method it exercises. For this, we exploit a traceability technique based on naming convention, *i.e.*, it identifies the methods under test by removing the string *'Test'* from the method name of the JUnit test method. This technique has been previously evaluated by Sneed [94] and by Van Rompaey and Demeyer [95], demonstrating the highest performance (both in terms of accuracy and scalability) with respect to other traceability approaches (*e.g.*, slicing-based approaches [96]).

Once we detect the links between test and production methods, we can compute the defect-proneness of such production methods. Since we calculate test smells at the release level (*i.e.*, we only have information regarding which test is smelly at the specific commit of the release), we have to detect how many defects production methods have within that particular release. To this aim, we rely again on the SZZ algorithm. To detect defects of production code in a specific release, we only consider bug fixing activities related to bugs introduced *before* the release date. More formally, we compute the fault-proneness of a production method M_i in a release r_j as the number of changes to M_i aimed at fixing a bug in the period between r_j and r_{j+1}, where the bug was introduced before the release date, in the period between r_{j-1} and r_j. The obtained list of bugs are the ones that were present in the system when it was released, hence not captured using tests.

By employing SZZ, we can approximate the time periods in which each production method was affected by one or more bugs. We exclude from our analysis all the bugs occurring in a production method M_i after the system was released, because in this case the test smell could have been solved before the introduction of the bug. We also exclude bug-introducing changes that were recorded after the bug was reported, since they represent false positives [97].

3.2.3 Data Analysis

To answer **RQ1**, we analyze the previously extracted information regarding test smells and change- and defect- proneness of test code. In particular, in the context of **RQ1.1**, we test whether JUnit test methods that contain a test smell are more likely to be change- or defect-prone. To this aim, we compute the Relative Risk (RR) [98], an index reporting the likelihood that a specific cause (in our case, the presence/absence of a test smell) leads to an increase in the amount a test case is subject to a particular property (in our case, number of changes or defects) [99, 100]. The RR is defined as the ratio of the probability of an event occurring in an exposed group (*e.g.*, the probability of smelly tests being defective), to the probability of the event occurring in a non-exposed group (*e.g.*, the probability of non-smelly tests being defective) and it is computed using the following equation:

$$RR = \frac{p_{\text{event when exposed}}}{p_{\text{event when not exposed}}}$$

A relative risk of 1 means that the event is equally likely in both samples. A RR greater than 1 indicates that the event is more likely in the first sample (*e.g.*, when the test is smelly), while a RR of less than 1 points out it is more likely in the second sample (*e.g.*, when the test is not smelly). We prefer using this technique rather than alternative statistical tests adopted in previous work (*e.g.*, analysis of box plots [85] or Odds

Ratios [84, 101]) because of the findings reported in the statistic field that showed how this method (i) should be preferred when performing exploratory studies such as the one conducted herein [102, 103] and (ii) is equivalent to Odds Ratios analysis [104].

Change- and defect-proneness of JUnit test methods might also be due to other factors rather than the presence of a test smell. Indeed, Kitchenham *et al.* [105] found that both size and number of previous changes might influence the observations on the defect-proneness of source code; additionally, Zhou *et al.* [106], reported the role of size as possible confounding effect when studying the change-proneness of code elements. Based on the evidence above, we control our findings for change-proneness by computing the RR achieved considering the size of the test method in terms of lines of code (LOC). Moreover, we control the phenomenon of defect-proneness by considering LOC of test methods and number of times the method changed from the last release (*i.e.*, prior changes). More specifically, the aim is to understand whether the likelihood of a test case being smelly *and* more change- or defect-prone varies when controlling for size and number of changes. In other words, if smelly tests are consistently more prone to changes and defects than non-smelly tests, independently from their size or number of times they changed in the past, we have higher confidence that the phenomena observed are associated with test smells.

To answer **RQ1.2** and analyze the role of test smell co-occurrences, we split the previously extracted dataset into seven groups, each one containing test methods affected by exactly i smells, where $0 \leq i \leq 6$. Then, we compare change- and defect-proneness of each group using (i) the Wilcoxon rank sum test [107] (with confidence level 95%) and (ii) Cohen's d [108] to estimate the magnitude of the observed difference. We choose the Wilcoxon test since it is a non-parametric test (it does not have any assumption on the underlying data distribution), while we interpret the results of Cohen's d relying on widely adopted guidelines [108]: The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$.

To answer **RQ1.3**, we adopt the same procedure as for **RQ1.2**, but we consider each smell type separately, *i.e.*, we compare change- and defect-proneness of different smell types by means of Wilcoxon rank sum test [107] and Cohen's d [108], controlling for size and number of previous changes (only in case of defect-proneness). It is important to note that, as done in earlier work [84, 85], in this analysis we consider test cases affected *only* by a single test smell, *e.g.*, only *Eager test*, with the aim of understanding the effect of single test smells on change- and fault-proneness of test code.

For **RQ2** we adopt a process similar to that of **RQ1**. In particular, for **RQ2.1** we compute the RR: in this case, we aim to investigate the likelihood that the presence/absence of a test smell is associated with the defect-proneness of the production code being tested. Similarly to **RQ1**, we control for size and number of changes. Analogously, in **RQ2.2** we use (i) the Wilcoxon rank sum test [107] and (ii) Cohen's d [109] to assess the association of test smell co-occurrences to the defect-proneness of production code. Finally, to answer **RQ2.3**, we compare the distribution of the number of defects related to the production code tested by different test smell types (considering *single* test smell types).

3.2.4 Threats to Validity

Our research method poses some threats to the validity of the results we obtain.

Construct validity. Threats to construct validity concern our research instruments.

To obtain information regarding test smells we use the test smell detector devised by Bavota *et al.* [11]. Even though this tool has been assessed in previous studies [11] as being extremely reliable, some false positives can still be present in our dataset.

Another threat is related to how we detected which production method is exercised by a test method: specifically, we exploited a traceability technique based on naming convention that has been heavily adopted in the past [12, 56, 86, 101]. This technique has also been evaluated by Sneed [94] and by Van Rompaey and Demeyer [95], and the results reported an average precision of 100% and a recall of 70%.

Internal validity. Threats to internal validity concern factors that could affect the variables and the relations being investigated. When we look into the relation between test smells and test defects, many factors can influence the results. For example, a test could contain more defects than others because more complex, bigger, or more coupled, while the studied variable (test smells) could be insignificant. To mitigate this, we control for some of these metrics, namely size of the method (LOC) and number of changes, which have been reported to correlate with code complexity [110]. As shown in the results section, the results generally do not change when controlling for other metrics. Furthermore, at the beginning of this study we also built a Logistic Regression Model to detect whether our explanatory variable was (not) statistically significant in the model. Similarly to Thongtanunam *et al.* [28], we built a logistic regression model to determine the likelihood of a test being defective (or change prone) using LOC, prior changes, production changes as control variables and being smelly (our new variable) as a binary explanatory variable. We used R scripts provided by Thongtanunam *et al.* [28] to build the model, and we discovered that test code smelliness was indeed statistically significant for the model. However, we preferred to proceed with RR instead of the model, for better readability of the results.

External validity. Threats to external validity concern the generalization of results. We conducted our study taking into account 221 releases of 10 Java systems having different scope and characteristics to strengthen the generalizability of our findings. However, a study investigating different projects and programming languages may lead to differing conclusions.

Table 3.2: Subject test smells

Test smell	Description	Problem
'Mystery Guest'	A test that uses external resources (e.g., file containing test data)	Lack of information makes it hard to understand. Moreover, using external resources introduces hidden dependencies: if someone deletes such a resource, tests start failing.
'Resource Optimism'	A test that makes optimistic assumptions about the state/existence of external resources	It can cause non-deterministic behavior in test outcomes. The situation where tests run fine at one time and fail miserably the other time.
'Eager Test'	A test method exercising more methods of the tested object	It is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.
'Assertion Roulette'	A test that contains several assertions with no explanation	If one of the assertions fails, you do not know which one it is.
'Indirect Testing'	A test that interacts with the object under test indirectly via another object	This smell indicates that there might be problems with data hiding in the production code.
'Sensitive Equality'	A test using the 'toString' method directly in assert statements	It may depend on many irrelevant details such as commas, quotes, spaces, etc. Whenever the toString method for an object is changed, tests start failing.

3.3 RQ1 Results: Test Smells and Test Code

This section describes the results to **RQ1**.

RQ1.1: To what extent are test smells associated with the change- and defect- proneness of test code?

Figure 3.1 depicts the Relative Risk of test smells to be associated with higher change-proneness of test cases (label "Overall") as well as how the risk is connected with the control factor analyzed, *i.e.*, size. In particular, we show how RR varies when the test method has (i) small size ($LOC < 30$), (ii) average size ($30 < LOC < 60$), and (iii) large size ($LOC > 60$). The thresholds used to identify small, medium, and large test methods were identified by applying the *Maintainability Model* proposed by Heitlager *et al.* [111], which cuts the distribution of all the method LOCs at the 70th, 80th and 90th percentiles. We also represent the *p*-value and the confidence interval for each category.

We make two main observations from the results in Figure 3.1. On the one hand, test methods affected by at least one smell are more change-prone than non-smelly methods, with an RR of 1.47; from a practical perspective, this means that a smelly test has the risk of being 47% more change-prone than a non-smelly test. On the other hand, we can notice that smelly tests with higher size are more change prone: this is intuitive since larger methods are more difficult to maintain (hence more change prone) and they are more likely to contain smells. An important result to notice is that large smelly tests ($LOC > 60$) are more than twice more likely of being change prone than not smelly large tests. This finding is a good incentive for practitioners and developers to write small and concise tests, as recommended by Beck [6].

Concerning defect-proneness, Figure 3.2 shows how the RR varies when considering (i) the presence of test smells ("Overall"), (ii) the size of test cases—split in the same way as done for change-proneness, and (iii) the number of previous changes applied to test cases (we discriminated between methods that change frequently vs. methods that infrequently change, by adopting the heuristic proposed by Romano and Pinzger [81], *i.e.*, we considered frequently evolving methods to have a number of changes higher than the median of the distribution of all the changes that occurred in test cases — 2, in our case).

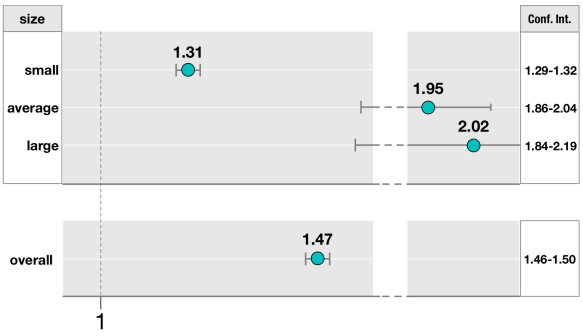


Figure 3.1: Relative risk of being change prone in smelly tests *vs* non-smelly tests, controlling by size. The p-value for all RRs is < 0.0001.

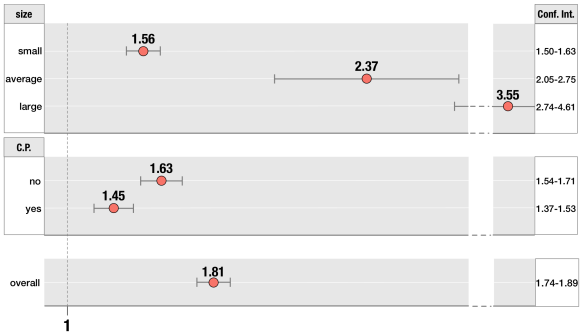


Figure 3.2: Relative risk of being defect prone in smelly tests *vs* non-smelly tests, controlling by size and change proneness. For all RRs, p-value < 0.0001.

From Figure 3.2, we observe that the presence of test smells is associated with the defect-proneness of test cases. Indeed, methods affected by at least one design flaw have the risk of being 81% more defect-prone than non-smelly ones. Additionally, the result does not change when controlling for size and number of changes. Indeed, the difference is even more prominent for large tests: the smelly ones are 3.5 times more defect prone than the not smelly. Instead, change proneness seems not relevant when discriminating the defect-proneness of test cases. In both cases, the RR of smelly tests of being more defect prone is 50% higher.

Overall, the results of this first analysis provide empirical evidence that test smells—defined with the aim of describing a set of bad patterns influencing test code maintainability [2]—are indeed associated with higher change- and defect-proneness of the affected test cases.

Summary: *Tests affected by test smells are associated with higher change- and defect-proneness than tests not affected by smells, also when controlling for both the test size and the number of previous changes.*

RQ1.2: Is the co-occurrence of test smells associated with the change- and defect-proneness of test code?

While in the previous research question we did not discriminate on the number of test smells a test method contained, the goal of this analysis is to assess whether test smell co-occurrences is associated with the change- and defect- proneness of test cases. Figures 3.3 and 3.4 report box plots showing change- and defect-proneness of test cases affected by a different number of test smells, respectively.

For change-proneness, the median of the different groups very low (around one) for all test cases: to some extent, this is in line with the findings by Zaidman *et al.* [76], who found that developers generally do not change test cases as soon as they implement new modifications to the corresponding production code. At the same time, Figure 3.3 shows that the higher the number of test smells, the more dispersed the distribution of changes

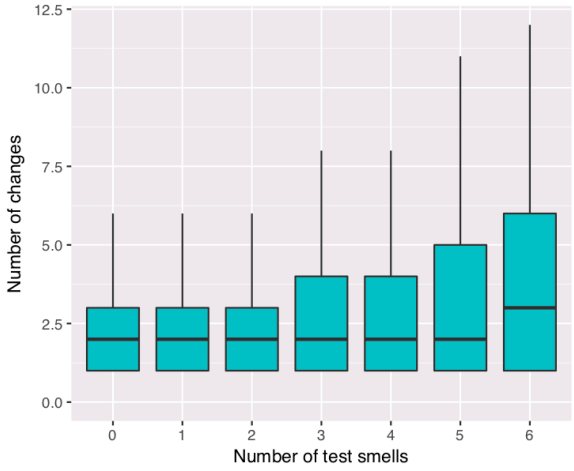


Figure 3.3: Number of smells in a test method and corresponding number of changes to the method.

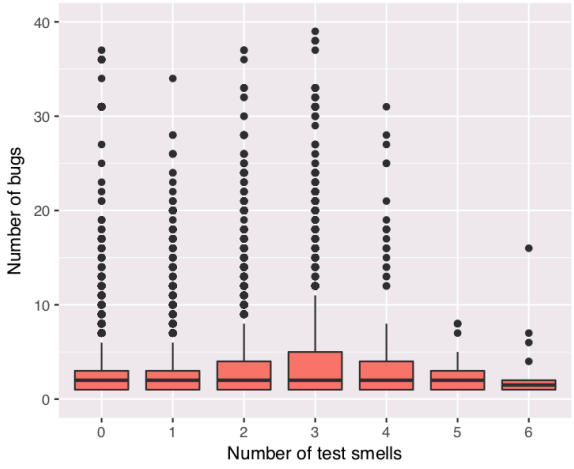


Figure 3.4: Number of smells and number of defects

is, thus indicating that test cases affected by more design problems tend to be changed more often by developers. This observation is supported by the results of the statistical tests, where we found that the difference between all groups was statistically significant ($p - value < 2^{e-16}$), with a negligible effect size between the first 5 groups ($d \leq 0.2$) and a medium one between the first 5 and the last 2 groups ($0.5 \leq d \leq 0.8$).

When considering defect-proneness in Figure 3.4, we notice that test methods having up to four test smells do not show significant differences with respect to methods affected by five or six design flaws. Indeed, the median of the distribution is almost identical in all the groups, and even though the difference is considered statistically significant by the Wilcoxon rank sum test, it has a small effect size ($d < 0.2$). Thus, these findings suggest

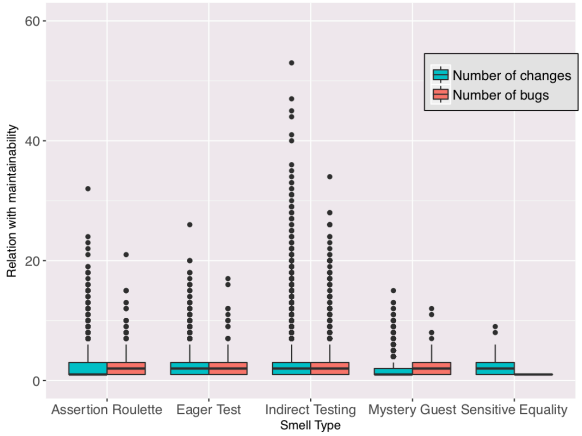


Figure 3.5: Change- and fault- proneness of test methods affected by different types of smells.

that the co-occurrence of more test smells is not directly associated with higher defect-proneness; we hypothesize that they are instead a co-existing phenomenon, similarly to what Palomba *et al.* reported for code smells in production code [85].

In the context of this research question, we controlled for the size of the test method and the number of its changes, finding that these factors are not associated with the investigated outcome. We include a report of this additional analysis in our on-line appendix [46].

> ***Summary:*** *Test methods affected by more smells are associated with a slightly higher change-proneness than methods with less smells. Conversely, the co-presence of more test smells in a test method is not associated with higher defect-proneness.*

RQ1.3: Are certain test smell types more associated with the change- and defect-proneness of test code?

The final step of the first research question investigates the association to change- and defect-proneness of different test smell types. Figure 3.5 shows two box plots for each type, depicting its change- and defect-proneness. When analyzing the change-proneness, we observe that almost all the test smells have a similar trend and indeed the magnitude of their differences is negligible, as reported by Cohen *d*. The only exception regards the *Indirect testing* smell: while the median change-proneness is similar to other smells, its box plot shows several outliers going up to 55 changes. In this case, the magnitude of the differences with all the other smell types is medium. This result is due to the characteristics of the smell. By definition, an *Indirect testing* smell is present when a method performs tests on other objects (*e.g.*, because of external references in the production code tested) [2]: as a consequence, it naturally triggers more changes since developers may need to modify the test code more often due to changes occurring in the exercised external production classes.

In the case of defect-proneness the discussion is similar. Indeed, the number of defects affecting the different test smell types is similar: even though the differences between them are statistically significant ($p\text{-}value < 2^{e-16}$), they are mostly negligible. However, we can see some exceptions, also in this case. The box plots show that the distribution of 'Indirect Testing', 'Eager Test' and 'Assertion Roulette' smells slightly differ from the others, and indeed these are the smells having the highest number of outliers. This result is due to the fact that these test smells tend to test more than required [2] (*i.e.*, a test method suffering from 'Indirect Testing' exercises other objects indirectly, an 'Eager Test' test method checks several methods of the object to be tested, while an 'Assertion Roulette' contains several assertions checking different behavior of the exercised production code). Their nature makes them intrinsically more complex to understand [11], likely leading developers to be more prone to introduce faults.

Summary: *Test methods affected by 'Indirect Testing', 'Eager Test', and 'Assertion Roulette' are more change and defect prone than those affected by other smells.*

3.4 RQ2 Results: Test Smells and Production Code

This section describes the results to our second research question.

RQ2.1: Are test smells associated with the defect-proneness of the tested production code?

Figure 3.6 reports the RR that a smelly test case is exercising a more defect-prone production method (label 'Overall'), along with the RR obtained when considering size as a control factor.

In the first place, Figure 3.6 shows that smelly tests have a higher likelihood to test defective code than non-smelly tests (*i.e.*, the RR = 1.59 states that production code executed by smelly tests has 59% higher chances of being defective than production code executed by non-smelly tests). Zooming in on this result, Figure 3.7 depicts the box plots reporting the distribution of the number of production code bugs, when exercised by smelly test

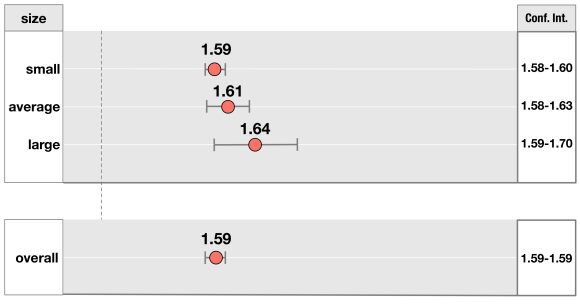


Figure 3.6: Relative risk of the production code being more defect prone when tested by smelly tests *vs.* non-smelly tests. For all RRs, p-value < 0.0001.

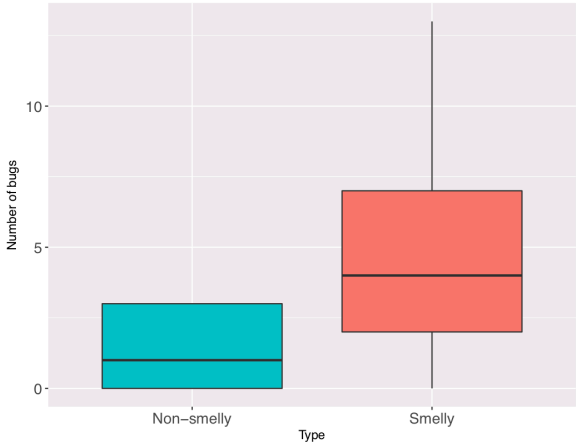


Figure 3.7: Distribution of the number of production code bugs when tested by smelly tests *vs* non-smelly tests.

methods vs. non-smelly ones. The difference between the two distributions is statistically significant ($p\text{-}value < 2.2^{e-16}$) with a large effect size ($d = 1.40$).

The results still hold when controlling for size: Size does not impact the RR concerning the defect-proneness of production code exercised by smelly tests vs. non-smelly ones, actually, as shown in the previous RQ, it makes it worst. For instance, methods having a large number of lines of code have an $RR = 1.64$. Two main factors can explain this result: On the one hand, we suppose that a large size of the test implies a large volume of the production code, and our research community widely recognized size as a valid proxy measure for software quality [105]; on the other hand, our results corroborate previous findings reported by Palomba *et al.* [85], who showed that large methods (*e.g.*, the ones affected by a *Long Method* code smell [78]) are strongly associated with the defect-proneness of production code.

Thus, from our analysis we have empirical evidence that the presence of test smells contributes to the explanation of the defect-proneness of production code. Given our experimental setting, we cannot speculate on the motivations behind the results achieved so far: indeed, our **RQ2.1** meant to be a coarse-grained investigation aimed at understanding whether the presence of design flaws in test code might somehow be associated with the defectiveness of production code. Thus, in this research question we did not focus on the reasons behind the relationship, *i.e.*, if it holds because the production code is of poor quality (thus difficult to test) or because the tests are of poor quality (thus they do not capture enough defects). Our **RQ2.3** makes a first step in providing additional insights on such a relationship.

> ***Summary:*** *Production code that is exercised by test code affected by test smells is more defect-prone, also when controlling for size.*

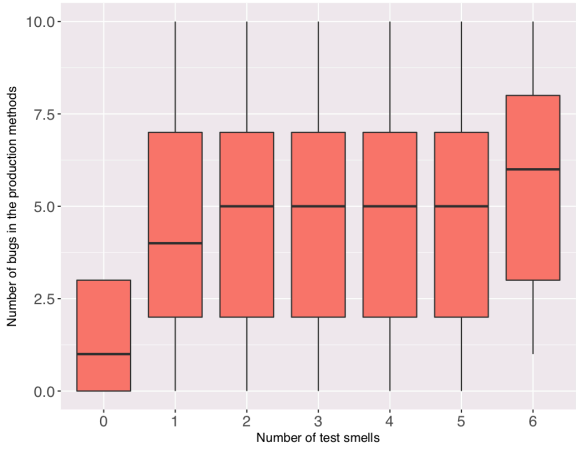


Figure 3.8: Number of smells and number of production defects.

RQ2.2: Is the co-occurrence of test smell associated with the defect-proneness of the tested production code?

Figure 3.8 presents the results concerning the association of test smell co-occurrences to the defectiveness of the exercised production code. In this case, the defect-proneness of production code remains almost constant among the different groups, meaning that having more design issues in test code is not associated with a higher number of defects in production.

This result led to two main observations: as observed in **RQ2.1**, test smells are related to the defect-proneness of the exercised production code, but do not fully explain this phenomenon. Secondly, while the specific number of test smells is not associated with the defectiveness of production code, the overall presence of test smells is. It is reasonable to think that some *specific* test smells could contribute more to the found association to defect-proneness; this reasoning represented the input for **RQ2.3**.

In this research question, we controlled the findings for size and number of changes, finding that none of them influence the outcome. We include a report of this additional analysis in our on-line appendix [46].

> ***Summary:*** *The co-occurrence of more test smells in a test case is not strongly associated with higher defect-proneness of the exercised production code.*

RQ2.3: Are certain test smell types more associated with the defect-proneness of production code?

Figure 3.9 depicts the box plots reporting the association of different test smell types to the defect-proneness of production code. We observed that the 'Indirect Testing' and 'Eager Test' smells are related to the production code being more defect-prone with respect to the other test smell types. The differences observed between the 'Indirect testing' and

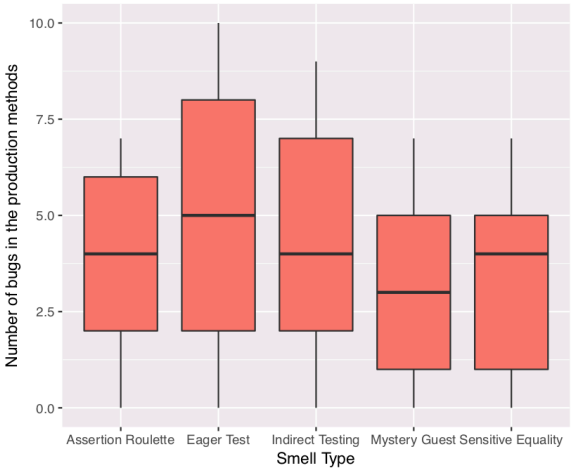


Figure 3.9: Number of defects for different types of smells.

'Eager Test' and the other distributions are all statistically significant ($p - value < 2^{e-16}$) with medium effect size, while we found the other smells to be not statistically associated with more production code defect-proneness.

As also explained in the context of **RQ1.3**, the 'Indirect Testing' and 'Eager Test' smells lead to test cases that are (i) less cohesive and (ii) poorly focused on the target production code [2]. The former implies the testing of other objects indirectly, the latter checks several production methods of the class under test. The *lack of focus* of such smells may explain why the corresponding production code is associated with defect-proneness: It seems reasonable to consider that the *greedy* nature of these two smells makes them less able to find defects in the exercised production code.

From a practical point of view, our results provide evidence that developers should carefully monitor test and production code involved with *Indirect Testing* and *Eager Test*. In fact, these are the smells that not only are related to more change- and defect-prone test code, but also to more defect-prone production code.

> ***Summary:*** *'Indirect Testing' and 'Eager Test' smells are associated with higher defect-proneness in the exercised production code. A likely motivation is the lack of focus of the tests on the target production code.*

3.5 Conclusion

Automated testing is nowadays considered to be an essential process for improving the quality of software systems [70, 71]. Unfortunately, past literature showed that test code can often be of low quality and may contain design flaws, also known as test smells [2, 11, 112]. In this chapter, we presented an investigation on the relation between six test smell types and test code change/defect proneness on a dataset of more than a million test cases.

Furthermore, we delved into the relation between smelly tests and defect-proneness of the exercised production code.

4

Investigating Severity Thresholds for Test Smells

Test smells are poor design decisions implemented in test code, which can have an impact on the effectiveness and maintainability of unit tests. Even though test smell detection tools exist, how to rank the severity of the detected smells is an open research topic. In this work, we aim at investigating the severity rating for four test smells and investigate their perceived impact on test suite maintainability by the developers. To accomplish this, we first analyzed 1,489 open-source projects to elicit severity thresholds for commonly found test smells. Then, we conducted a study with developers to evaluate our thresholds. We found that (1) current detection rules for certain test smells are considered as too strict by the developers and (2) our newly defined severity thresholds are in line with the participants' perception of how test smells have an impact on the maintainability of a test suite.

Preprint: https://doi.org/10.5281/zenodo.3744281,

Data and material: https://doi.org/10.5281/zenodo.3611111.

This chapter has been published as *Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. Investigating Severity Thresholds for Test Smells. In Proceedings of the 17th International Conference on Mining Software Repositories – MSR '20, pages 311–321, New York, NY, USA, June 2020. ACM.* [113]

4.1 Introduction

Violations of design principles (*a.k.a.*, code smells) are not restricted to production code, but are also found in (unit) test code [2, 7, 78]. Such *test smells* can lead to harder to maintain tests [11, 69, 114], just as (production) code smells can increase maintenance effort [115].

Developers tend to focus on production code quality, while test code quality is often not prioritized [69]; moreover, once test smells are introduced, they are hardly ever removed through refactoring [12]. One could argue that the concept of test code quality and test smells in particular is in need of further investigation. For example, previous research has reported that developers do not always perceive test smells as problematic [12], but the actual reason is unclear. One reasonable explanation is that current test smell detection tools lack *severity thresholds*, which could make their indications more actionable. Indeed, Alves *et al.* showed the importance of determining severity thresholds for (production) code smells to "adequately support subsequent decision-making" [116], by successfully using their defined thresholds for software analysis, benchmarking, and certification in industry [116]. In this study, we investigate severity thresholds for test smells.

Particularly, our aim is to:

- Calibrate detection thresholds such that severity levels can be assigned to a test smell instance, allowing developers to focus on the higher severity smells.
- Improve the accessibility of automatic test smell detection by integrating into existing developer tooling.

In their work, Alves *et al.* defined a method to calibrate thresholds for code quality metrics, which they named as 'benchmark based threshold derivation methodology' [116]. Their approach consisted of collecting data from existing software systems and using the distributions of metric values to find appropriate thresholds [83, 116]. In our investigation, we take a similar approach. We collect the (unit) test code of 1,489 Java projects from the Apache and Eclipse ecosystems and apply the open-source test smells detection tool TSDETECT [117]; then, we use these systems as a benchmark to derive the values for three severity thresholds: 'Medium', 'High', 'Very high'. Following this approach, we found that four of nine test smells we considered should have higher thresholds than what previously reported in literature [69, 77, 114].

Subsequently, we move to our second goal: We integrate test smell detection provided by TSDETECT into a prototype (back-end) extension of BETTERCODEHUB [44] (BCH), a web-based code quality analysis tool provided by SIG [1]. We engaged the existing users of BCH to interact with the new test smell prototype and solicited their feedback on instances of test smells within their own code bases. A total of 31 developers, across 47 diverse projects, answered, providing data points on 301 detected test smells. These responses allowed us to evaluate the developers' perceptions of our newly proposed thresholds. According to the developers, Empty Test, Sleepy Test, and Mystery Guest have the highest priority as refactoring candidates, while Empty Test, Ignored Test, and Conditional Test Logic are considered the smells with the higher impact on code maintainability. Furthermore, the ratings submitted by the users are aligned with our thresholds, with a statisti-

[1] https://www.softwareimprovementgroup.com

cally significant difference and a strong Spearman's coefficient, suggesting that the newly defined thresholds can be used to prioritize test smells instances.

Our study makes the following contributions:

1. Calibrated severity thresholds for test smell detectors
2. A mechanism for rating the severity of certain test smells, allowing tools to classify test smells into distinct categories based on their severity. This enables developers to only focus on the most critical test smells.
3. An integration of an automatic test smell detector within a GitHub-based code quality tool (BCH).
4. An evaluation of developers' perceptions of test smells within their own code bases, by integrating our new thresholds into a GitHub-based code quality tool (BCH), thus validating our test smell severity levels.

4.2 Related work

4.2.1 Test Smell Detection Tools

Van Rompaey *et al.* created a metrics-based test smell detection tool for Java to detect General Fixtures and Eager Tests [10, 118]. Later Breugelmans and van Rompaey developed the TestQ tool, which works with C, C++ and Java test code to detect the following test smells: Assertion Roulette, Eager Test, Empty Test, For Testers Only, General Fixture, Indented Test (the equivalent of Conditional Test Logic), Indirect Test, Mystery Guest, Sensitive Equality and Verbose Test [119].

Greiler *et al.* focused on identifying common problems with test fixtures. They implemented a tool called TestHound, which works on JVM bytecode level and can detect the following test smells: General Fixture, Test Maverick, Dead Field, Lack of Cohesion of Test Methods, Obscure In-Line Setup and Vague Header. In this tool, they showed the code impacted by test smells to the developers along with tips for how to refactor the test code [8]. This setup is comparable to the one we implemented in BCH, as we also present code instances impacted by test smells. They concluded that having a tool to point out the problems with the test suite can help the developers with refactoring.

Palomba *et al.* developed TASTE (Textual AnalySis for Test smEll detection), which can detect General Fixtures, Eager Tests, and Lack of Cohesion of Methods using Information Retrieval techniques, thus bypassing the need to fully parse the test code. Their tool shows a better precision and recall than the AST-based tools TestQ and TestHound [77].

Satter *et al.* [120] created a tool for the detection of dead fields in Java unit tests. They report a better detection accuracy than TestHound.

Bavota *et al.* studied the diffusion of test smells in open-source and industrial projects [11]. To help with their research, they created a test smell detection tool for Resource Optimism, Indirect Testing, Test Run War, Mystery Guest, General Fixture, Eager Test, Lazy Test, Assertion Roulette, For Testers Only, Test Code Duplication and Sensitive Equality.

Zhang *et al.* focused on dependencies between tests by empirically investigating issue tracking systems. They developed a tool which identifies dependencies between tests on

a test suite level, not on individual test case level). Their approach requires executing the tests (dynamic analysis) [121].

Gambi *et al.* also looked into test dependency detection and developed the tool PraDeT, which also requires running the tests to find dependencies [122].

4.2.2 How Developers Perceive Test Smells

Tufano *et al.* asked 19 developers from various open-source projects to look at test code samples which contained instances of test smells. In the majority of the presented cases (82%), the developers did not recognize any problems with the test code. The code presented was created or maintained by the interviewed developers and thus the developers saw or created the presented code before the interview session. The authors highlight the need for having automated tools that can detect test smells and present them to the developers. They have also found out that the majority of test smells are created during the initial test development and are not removed in subsequent refactoring [12].

Palomba *et al.* investigated the developer's perception of code smells, using both the original authors and independent developers. They asked the developers to identify the design problems (code smells) and, if found, give them a severity rating. The severity ratings were applied to the whole code smell category, and not to the specific code smells instances. Furthermore, the research focused only on production code smells and not test smells. They have then split the observed code smells into 3 categories: Generally not Perceived as Design or Implementation Problems, Generally Perceived and Identified by Respondents, and Perception may Vary [59].

Both of these studies differ from ours in execution, as they presented developers test code sections without further context, and outside of the developer's usual work environments. In contrast, our study asks developers to work within the normal work-flow of their code quality tool (BCH) and within the context of their own project.

Kummer studied whether developers recognize test smells using a sample of 20 developers. The author concludes that test smells can be refactored by the developers without them knowing that they are specific instances of test smells and suggests that automated test smell detection tools could help the developers further justify their removal [123].

4.3 Methodology

Our overall research goal is to investigate severity thresholds for test smells. The most common test smell detection tools work on a binary scale of whether a given code is impacted by a test smell or not without any additional data, thus ignoring commonly encountered situations in software development.

In our first research question, we seek to define new thresholds for test smells, based on the 'benchmark based threshold derivation methodology' [116] considering a large number of software systems.

RQ$_1$. *How can test smells be given a severity rating?*

In our second research question, we aim to challenge our newly defined thresholds with users, therefore we ask:

RQ$_2$. *What is the perception of developers on test smells in their codebase?*

4.3.1 Test Smells Tool Selection

We researched the available test smell detection tools, detailed in Section 4.2.1. Given that most of the better validated tools only support Java for the widest variety of test smells, we decided to focus on test smell research in Java. We selected the open-source tool tsDetect [117] for finding test smells in the codebase. The tool works with Java JUnit projects and has support for JUnit 4 annotations, which can be extended to support JUnit 5, and has precision and recall above 85% [117]. It also has a published accuracy rating along with manually classified test smell data [117], providing a test suite for the tool extension and testing. The main advantages of tsDetect are that it is open-source, developed recently in 2018, uses AST-based detection of test smells, and supports adding new test smells and detection rules. Compared to other tools, the support for JUnit along with using only AST information instead of text search for pattern violations results in improved accuracy.

4.3.2 Test Smells Selection

tsDetect can detect multiple test smells. However, for this study we restricted our analysis to the test smells depicted in Table 4.1. We select these test smells because they do not require viewing the full test source code to understand whether they are smelly or not. Certain test smells, such as Lazy Test (multiple test methods invoking the same method of the production object), require viewing the entire test class code to evaluate, while in this research we are focused on detecting test smells at method level. Based on testing done on a selected dataset, tsDetect is highly reliable at detecting instances of these test smells [117]. Previous studies [114, 124] report the distributions of the smells we analyzed along with others we did not select. The F-Score for tsDetect on the selected subset of test smells is between 87% and 99% [117].

Table 4.1: Subject test smells

Test smell	Description	Problem
'Mystery Guest'	A test that uses external resources (e.g., file containing test data) [2]	Lack of information makes it hard to understand. Moreover, using external resources introduces hidden dependencies: if someone deletes such a resource, tests start failing.
'Resource Optimism'	A test that makes optimistic assumptions about the state/existence of external resources [2]	It can cause non-deterministic behavior in test outcomes. The situation where tests run fine at one time and fail miserably the other time.

...continued on next page

Table 4.1 – continued from previous page

Test smell	Description	Problem
'Eager Test'	A test that tries to verify too many functionalities, which can lead to difficulty in understanding the test code [2]	It is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.
'Assertion Roulette'	A test that has multiple assertion statements that do not provide any description of why they failed [2]	When the test fails, investigating the reason can be problematic as the assertion statement might not provide the reason why the test could have failed.
'Conditional Test Logic'	A test that has control flow statements inside a test [7]	Tests can have multiple branch points and greater care must be taken when analyzing whether the test is correct.
'General Fixture'	It occurs when the test setup method creates fixtures (class fields used by the test cases) and a portion of the tests use only a subset of the fixtures [2]	The presence of this smell can result in longer execution of tests, as tests that do not use all of the fixtures still need to wait for the creation of all test fixtures.
'Empty Test'	It occurs when a test method does not contain executable statements	An empty test can be considered problematic and more dangerous than not having a test case at all since JUnit will indicate that the test passes even if there are no executable statements present in the method body.
'Magic Number Test'	A test that contains so-called "magic numbers", which are numbers used in assertions without any explanation where they come from and their value may not be immediately clear from just looking at the test code [7, 125]	The magic numbers should be replaced by a named constant, where the name describes where the value comes from or what it represents.
'Sleepy Test'	A test which contains thread suspension calls [7]	The use of this method call introduces additional delay to the test execution.
'Verbose Test'	A test that is too long and hard to understand [7]	Too long tests prevent them from being used as documentation and they are harder to maintain due to their complexity.

...continued on next page

Table 4.1 – continued from previous page

Test smell	Description	Problem
'Ignore Test'	A test method or class that contains the @Ignore annotation	ignored test methods result in overhead since they add unnecessary overhead with regards to compilation time, and increases code complexity and comprehension.

4.3.3 Preliminary Study

Before running the study in large scale, we perform a preliminary study within SIG. In this phase, we test the integration of tsDetect (without any modification) in BetterCodeHub.

During the pilots, the developers were asked to answer three questions: (1) whether the test smell instance is valid in the project context, (2) whether they classify the smell as "refactoring candidate" or as something which will take too much time and effort to fix (technical debt), and (3) rate from 1 to 5 the importance of the test smell instance on the project's maintainability. After completing the pilot, the developers were interviewed about their experience and asked for additional feedback, so that we could implement them before running the actual study. Four developers of SIG participated in the pilot, and each of them was asked to evaluate parts of the codebase written in Java they were actively working on as part of their daily job while using our prototype for the evaluation.

The main result of this preliminary study is that many of the test smell instances were rated as false positive or no-fix. The main reason is that in many of these instances developers did not see a design issue. For example, for the "Conditional Test Logic" smell developers complained that having only one branch (*e.g.*, *if-else*, or a *for*) should not be considered as a high priority refactoring. Similarly, developers did not agree on how "Eager Test" is calculated: currently, a method suffers from this smell if it contains more than one production call. However, developers agreed that almost all the tests have at least two production calls, hence resulting in many test methods rated as Eager Tests even though they are not.

On the other hand, in some cases developers acknowledged the problem and were willing to refactor the test method. However, since there is no distinction between having one production call or 10, these cases were mixed together with the false positives.

The main takeaway of this preliminary study is that test smells presented to the developers should be prioritized. The prioritization could help them focus only on the most severe test smell instances first, before moving onto fixing the less severe ones. The current implementation of test smell detection does not take into account additional information about the test smells, such as what percentage of the test code it impacts or how severe it is. Presenting the developers with test smells they perceive to be of low severity could lead them being more likely to ignore them and also trust the tool less. For these reasons, we need to determine a way to prioritize test smells, by means of new severity thresholds.

4.3.4 RQ1: Defining Severity Thresholds

With RQ1 we aim at creating new severity ratings for each test smell. Hence, we first define metrics for each test smell, based on their definitions and characteristics as proposed by van Deursen *et al.* [2] and Meszaros [7]. The metrics are shown in Table 4.2. Here the metrics denote the value which will be collected for each test method. Due to the binary nature of Empty Test and Ignored Test, we excluded them from the calibration metrics, and all instances of these smells are classified as the highest severity.

Table 4.2: Metrics used for threshold derivation, by test smell

Test Smell	Metric
Assertion Roulette	# assertions without description
Conditional Test Logic	# conditional statements
Eager Test	# production method calls
General Fixture	# unused fixtures
Magic Number Test	# magic numbers
Mystery Guest	# external files used
Resource Optimism	# files not checked for existence
Sleepy Test	# thread suspend calls
Verbose Test	# statements

Once the metrics are defined, we need to give them a severity rating. For this, we used the Benchmark-based threshold derivation proposed by Alves *et al.* [116] methodology, which follows three core principles [126], which states that the method should (1) be driven by measurement data from a representative set of systems (data-driven), rather than expert opinion, (2) respect the statistical properties of the metric (*e.g.*, the metric scale and distribution), (3) be resilient against outliers in metric values and system size (robust), (4) be repeatable, transparent, and straightforward to carry out (pragmatic). We chose this technique as it (i) does not assume the normality of the metric values distribution, (ii) uses a weight function (LOC), which emphasizes the metric variability, (iii) separates the thresholds into different risk categories. Furthermore, this state-of-art benchmarking technique has been used in many previous studies that needed to calculate thresholds for new metrics [69, 126–129].

The Benchmark-based threshold derivation enables us to define severity levels based on the representation of occurrences in the benchmark dataset.

Dataset. To apply this technique and derive the thresholds, we need a large and representative enough dataset. For this purpose, we selected *all* the projects from the Apache Software Foundation[2] and the Eclipse Foundation[3], which contained Java code. A total of 1,489 projects were selected. We use these projects because (1) the source code is publicly available, and (2) systems have different sizes and scopes. This will serve as our dataset. The dataset has a total of 25,356,827 LOC (project average: 31,617, median: 6,650).

Thresholds calculation. After the first step of selecting the projects, we follow the Benchmark-based threshold derivation methodology [116], consisting of the following 6

[2]https://www.apache.org/
[3]https://www.eclipse.org/

steps:

1. **Metrics extraction**: for each test method in the systems, we extract test smell metric information (see Table 4.2) and LOC to be used as weight.
2. **Weight ratio calculation**: for each method, we compute the weight percentage within its system, *i.e.*, we divide the method weight by the sum of all weights of the same system.
3. **Entity aggregation**: we aggregate the weights of all methods
4. **System aggregation**: we normalize the weights for the number of systems and then aggregate the weight for all systems.
5. **Weight ratio aggregation**: we order the metric values in ascending order.
6. **Thresholds Derivation**: we find the 70-80-90 percentiles to determine the thresholds.

After repeating this procedure for each test smell, we obtain 3 thresholds per smell for which we have defined a metric and weight. For instance, to represent 90% of the overall code for the Eager Test metric, the derived threshold is 39. In other words, we can say that 90% of the test methods (weighted by LOC and system size) have less than 39 production calls. This threshold is meaningful, since not only it means that it represents 90% of the code of a benchmark of systems, but it also can be used to identify 10% of the worst code [116]. To notice that the old threshold for this specific smell was the value one. We chose the percentile ranges of 70, 80, and 90 for each test smell, as this represents the increasingly worse portion of the codebase by volume [69, 83, 116] and thus represents the severity.

The thresholds allow the classification of test smells in a codebase into 4 distinct categories: test smell not found or below threshold (percentile intervals [0,70)), medium severity test smell for long term refactoring (percentile intervals [70,80)), high severity test smell for short term refactoring (percentile intervals [80,90)) and very high severity test smell for immediate refactoring (percentile intervals [90,100]).

4.3.5 RQ2: Developers' perceptions

To study the perception of developers on test smells found in their codebase, based on our thresholds, we modified BCH to present found test smells to the developers who could then provide their insights. By using the developer's repositories, we avoid the problem of developers seeing the code without knowing the project's context. Furthermore, we perform an analysis of test smells that were not considered for removal by the developers.

User Interface Design. Following the results of the new thresholds derived in RQ1, we modify BCH to integrate the test smells detector with the new thresholds. The front-end was also modified to show the analysis results along with our survey to gather developer feedback for each found test smell. This allows the developers who use BCH to access the results as part of their regular workflow, whether as a one-off analysis or continuous monitoring during pull requests or on a per commit level. Figure 4.1 shows an example of how the result was presented to the user.

Clicking on the test smell instance brings the users to the screen in Figure 4.2, where they can see the source code of the problematic code along with the survey. The users also had the ability to submit a GitHub issue to their repository with a description of the test smell and how it could be removed. The top displays a short explanation of what the test smell is and why it is harmful with a link to a more detailed page containing examples and explanations.

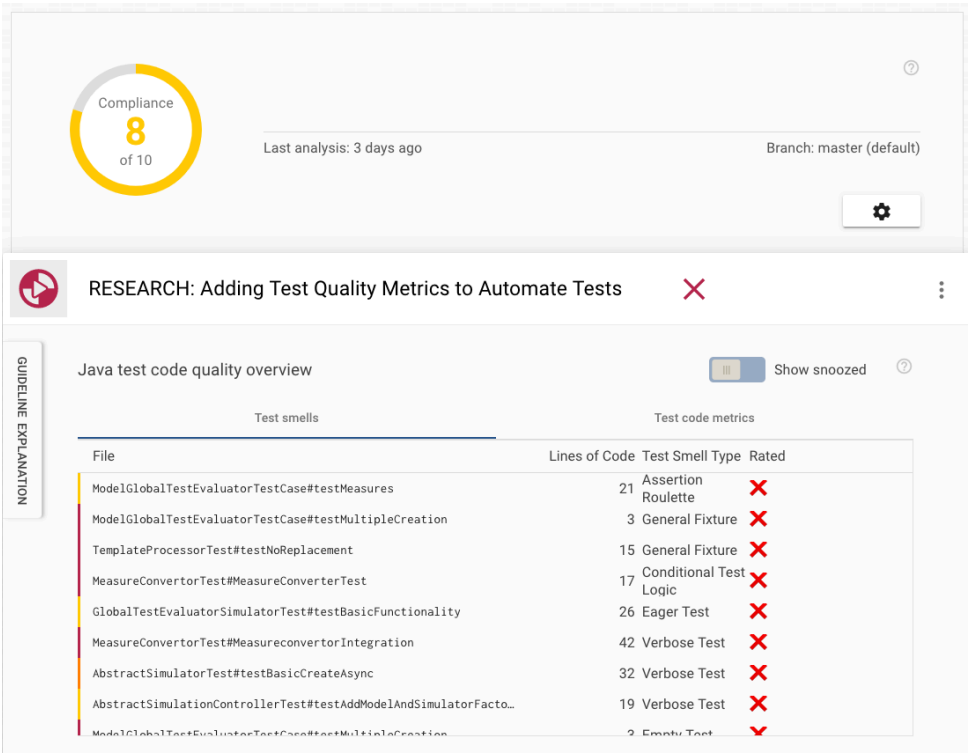


Figure 4.1: BCH view of test smell guidelines and violations.

Questions' Design. The questions asked to users about each found test smell are the following:

SQ1 Please rate the impact of this test smell instance on the test suite maintainability. *Options:* Don't know, False Positive, Very Low, Low, Medium, High, Very High.

SQ2 What action will you take with this test smell? *Options:* Mark for refactoring as soon as possible; Mark as technical debt for later refactoring; Ignore as won't fix / false positive.

SQ3 Estimated effort to refactor in person-hours. *Options:* numeric value (optional)

SQ4 Additional remarks. *Options:* free text (optional)

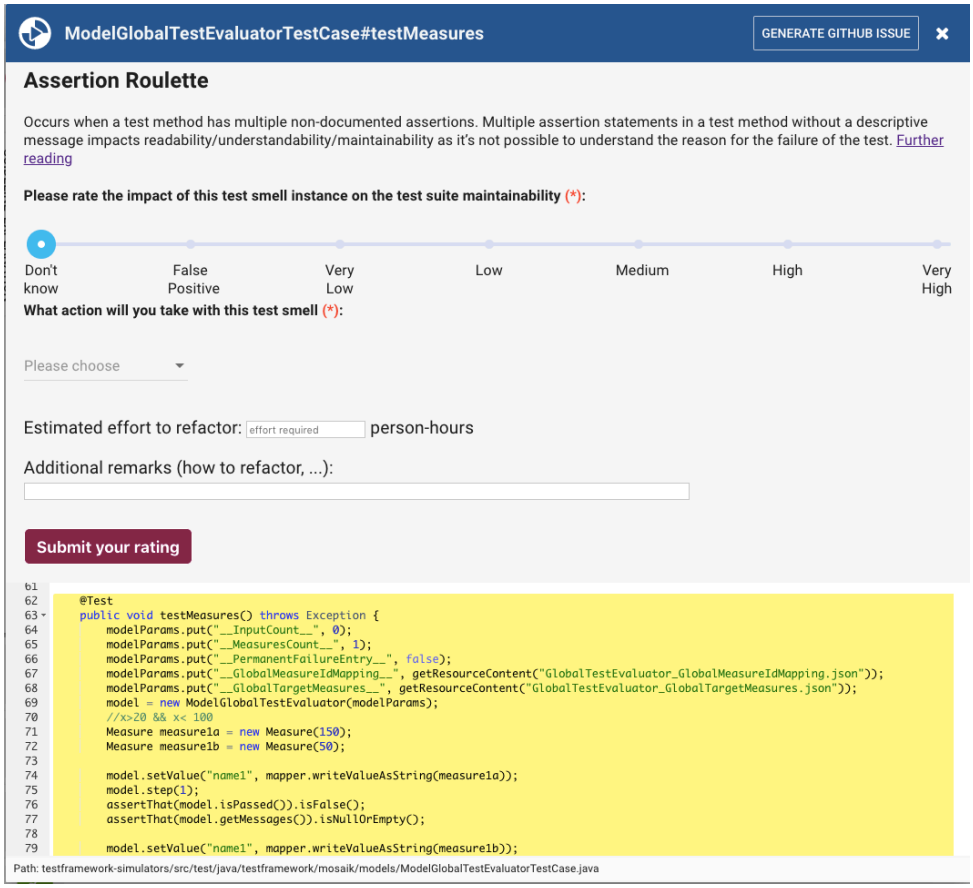


Figure 4.2: BCH details on a found test smell with the corresponding survey form.

To avoid duplication, the questions (Figure 4.2) are only displayed if the user has not submitted a rating for the specific smell instance.

Attracting Participants. We sent an email to all the BCH users ($N \approx 13,000$) to invite them to test these new changes on a dedicated research server. The email also contained a brief text and video guideline on how to use these new features. A total of 31 users responded, providing feedback on more than 300 test smell instances, which have been all analyzed.

4.4 Results

In this section we present our results by research question.

4.4.1 RQ1: Severity Thresholds

We analyzed a corpus of 1,489 projects to determine the severity thresholds for 9 test smells (Empty Test and Ignore Test are excluded from this calibration since they are binary by default). The calibration results are shown in Table 4.3.

As an example, Assertion Roulette has medium, high and very high severity set to 3, 5 and 10 respectively. These numbers represent the metric value (see Table 4.2): in the case of Assertion Roulette, it represents the number of assertions without descriptions. This means that if a method has a test smell value below 3, it should be considered not smelly (or in other words, it belongs to the best 70% of the corpus); if it has a test smell value between 5 and 10 it should be considered as high severity (it belongs to the worst 20% methods of the corpus); above a value of 10 it should be considered as very high severity, since it belongs to the worst 10% of the corpus.

As we can notice, 5 test smells have severity equals to 0: this means that the metric by which the smell is measured is so rare that even by considering the 90_{th} percentile we obtain a severity value of 0. Hence, for these cases, any test having a metric value higher than 1 results in a classification of very high severity. The distribution of the metrics across the observed projects for each test smell can be found in the replication package [47].

Table 4.3: Severity thresholds calculated from the benchmark

Test Smell	Severity Threshold		
	Medium	High	Very High
Assertion Roulette	3	5	10
Eager Test	4	7	39
Verbose Test	13	19	30
Conditional Test Logic	0	1	2
Magic Number Test	0	0	1
General Fixture	0	0	0
Mystery Guest	0	0	0
Resource Optimism	0	0	0
Sleepy Test	0	0	0

Previous studies have empirically investigated to what extent test smells are spread in software systems, by analyzing the distribution of test smells in source code [114, 130].

However, since our derived thresholds are higher, the diffusion of test smells would decrease.

As a first step to understand the test smells diffusion in open source projects using the new derived thresholds, we re-run the test smell analysis on our corpus of 1,489 projects using the new aforementioned thresholds. In Figure 4.3 we present the result. As we might expect since the new thresholds are stricter, we obtain from 8% to 30% less test smells instances than when using the old thresholds. To notice that the diffusion of the old thresholds are in line with what previous studies found [114]: "Assertion Roulette" is present in ≈ 50% of the test classes and "Eager Test" in 30% of them. However, by applying the new thresholds we obtain that "Assertion Roulette" is present in ≈ 30% of the classes and "Eager Test" in ≈ 15% of them. As for "Conditional Test Logic" and "Verbose Test" we could not find previous literature on their diffusion, however from the figure we can see that their occurrence is much lower when using the new thresholds.

In Table 4.4 we present the result for every test smell. Given that we defined new thresholds for only four smells, the numbers for the others are identical. Since out of the scope of this chapter, we did not further investigate the test smells diffusion on OSS systems. New research should be carried on this topic, for example by investigating the co-occurrence of the smells, similar to what previous studies did with the old thresholds [114].

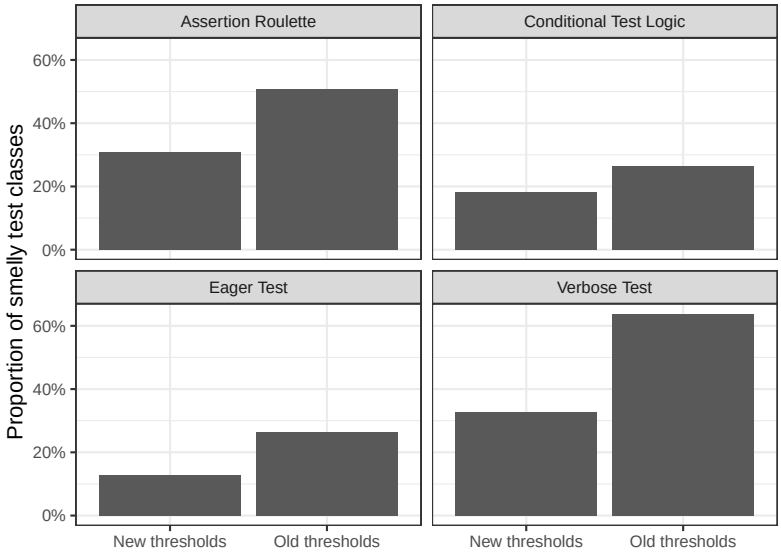


Figure 4.3: Diffusion of test smells across OSS systems: new thresholds *vs* old thresholds (N=1,489)

Thresholds' evaluation. After identifying the new thresholds, we want to determine if they are also aligned with the developer's perception of how much impact these instances have on the test suite maintainability. To this aim, we modify tsDetect to incorporate the defined thresholds and plug it in BCH. Using a survey (explained in Section 4.3.5), we ask the developers their perception of the shown test smell severity by using a Likert Scale [131]. Then we triangulated our severity score to the users' perception of severity

Table 4.4: Test smells' distribution across systems (N=1,489)

Test Smell	Old Thresholds		Derived Thresholds	
	Impacted	Not impacted	Impacted	Not impacted
Assertion Roulette	56,177 (50.9%)	54,278 (49.1%)	34,156 (30.9%)	76,299 (69.1%)
Cond. Test Logic	29,077 (26.3%)	81,378 (73.7%)	20,095 (18.2%)	90,360 (81.8%)
Empty Test	1,280 (1.2%)	109,175 (98.8%)	1,280 (1.2%)	109,175 (98.8%)
General Fixture	8,829 (8.0%)	101,626 (92.0%)	8,829 (8.0%)	101,626 (92.0%)
Mystery Guest	6,071 (5.5%)	104,384 (94.5%)	6,071 (5.5%)	104,384 (94.5%)
Sleepy Test	4,500 (4.1%)	105,955 (95.9%)	4,500 (4.1%)	105,955 (95.9%)
Eager Test	29,333 (26.6%)	81,122 (73.4%)	14,284 (12.9%)	96,171 (87.1%)
Ignored Test	3,105 (2.8%)	107,350 (97.2%)	3,105 (2.8%)	107,350 (97.2%)
Resource Optimism	7,345 (6.6%)	103,110 (93.4%)	7,345 (6.6%)	103,110 (93.4%)
Magic Num. Test	18,920 (17.1%)	91,535 (82.9%)	18,920 (17.1%)	91,535 (82.9%)
Verbose Test	70,461 (63.8%)	39,994 (36.2%)	36,080 (32.7%)	74,375 (67.3%)

employing the Spearman's rank correlation coefficient. We use the Spearman's rank correlation test, as we could not make any assumptions about the distribution of our data, thus ruling out the use of Pearson's test [132].

In Table 4.5 we show the results: all the four test smells for which we defined non-binary severity thresholds have a statistically significant difference between our proposed severity and user-perceived maintainability impact. Furthermore, Verbose Test and Conditional Test Logic showed a high statistically significant relationship ($p < 0.001$) and a strong Spearman's coefficient ($0.6 \leq r_s \leq 0.79$), while Eager Test and Assertion Roulette showed a lower statistically significant relationship ($p < 0.05$) and a weaker Spearman's coefficient ($0.2 \leq r_s \leq 0.39$).

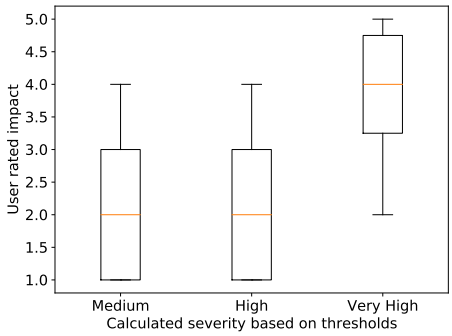
Table 4.5: Spearman's rank correlation between test smell severities as set by our thresholds and rated by users. Statistically significant results are in bold.

Test Smell	Responses	p	r_s
Eager Test	42	**0.027**	0.342
Conditional Test Logic	36	**<0.001**	0.753
Verbose Test	51	**<0.001**	0.679
Assertion Roulette	47	**0.016**	0.350

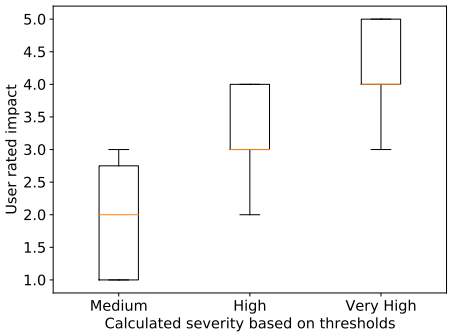
Figure 4.4 shows the distribution of the user-submitted impact ratings for each test smell severity category. As previously discussed, we can notice that for Verbose Test and Conditional Test Logic, the ratings submitted by the users are aligned with our thresholds, and the relationship is strong. For Eager Test and Assertion Roulette the difference is less visible instead, though statistically significant. Across the four smells, the threshold that aligns better with developers' perceptions is 'Very High', thus suggesting that it is the most appropriate to be used with practitioners.

4.4.2 RQ2: Developers' Perceptions

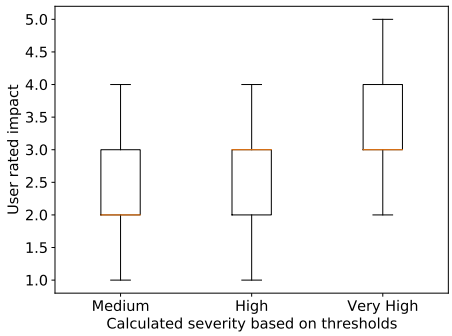
In this RQ, we want to investigate developers' overall perception on test smells found in *their* codebase: to this aim, we asked them to indicate for each test smell instance whether



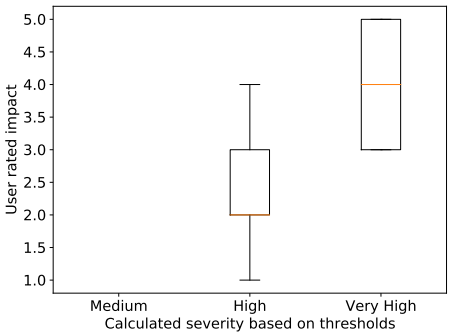
(a) Eager Test



(b) Verbose Test



(c) Assertion Roulette



(d) Conditional Test Logic

Figure 4.4: Test smell severity vs. user rated impact values

they would classify it as a valid instance to remove, how much priority they would give to the refactoring, and how long it would take according to them.

Refactoring evaluation. In Table 4.6 we show the percentage of actions the surveyed users indicated to take for each test smell instance identified in the BCH analysis report. We got a total of 301 responses. In Table 4.7 we show the impact ratings for test smell instances which were rated as either short or long term refactoring candidates (no action excluded).

Table 4.6: Actions taken by users, by identified test smell

Test Smell	Responses	Immediate Refactor	Long-term Refactor	No Action
Empty Test	14	71.43%	21.43%	7.14%
Sleepy Test	13	61.54%	23.08%	15.38%
Mystery Guest	10	60.00%	10.00%	30.00%
Resource Optimism	12	58.33%	16.67%	25.00%
Cond. Test Logic	39	53.85%	33.33%	12.82%
Verbose Test	54	48.15%	40.74%	11.11%
Magic Number Test	23	47.83%	34.78%	17.39%
Assertion Roulette	55	29.09%	50.91%	20.00%
Eager Test	51	21.57%	50.98%	27.45%
Ignored Test	10	20.00%	50.00%	30.00%
General Fixture	20	20.00%	40.00%	40.00%

Table 4.7: Impact ratings on test suite maintainability

Test Smell	Responses	Very Low	Low	Medium	High	Very High
Assertion Roulette	47	17.0%	27.7%	38.3%	14.9%	2.1%
Conditional Test Logic	36	2.8%	25.0%	30.6%	22.2%	19.4%
Eager Test	42	31.0%	26.2%	21.4%	16.7%	4.8%
Empty Test	13	0.0%	15.4%	7.7%	46.2%	30.8%
General Fixture	13	23.1%	30.8%	38.5%	7.7%	0.0%
Ignored Test	8	12.5%	12.5%	12.5%	37.5%	25.0%
Magic Number Test	20	15.0%	20.0%	30.0%	30.0%	5.0%
Mystery Guest	8	0.0%	12.5%	37.5%	50.0%	0.0%
Resource Optimism	10	10.0%	0.0%	20.0%	60.0%	10.0%
Sleepy Test	11	0.0%	9.1%	18.2%	63.6%	9.1%
Verbose Test	51	13.7%	23.5%	23.5%	27.5%	11.8%

Smell evaluation. We now discuss the finding for each test smell separately, taking into account the optional additional remarks field (**SQ4**) of the survey, where the user could add remarks regarding their perception of the smell.

Assertion Roulette: Assertion Roulette is rated as having a low to medium impact on test suite severity and considered primarily as long term technical debt. One reason to mark assertion roulette as a no action item is that the test method name accurately reflects the reason for the test to fail and thus no further comments in the assertions are required. In some instances, the purpose of a block of assertions was described in a

comment, which would require looking at the source code and line number to see why the test failed.

Conditional Test Logic Conditional Test Logic was primarily rated as a short-term refactoring candidate having medium to high impact. Of the 39 reported cases, 24 times it was reported as immediate refactoring, 13 as long-term refactoring and 2 times as no actions. In the comments, some developers stated that they use conditions to avoid running the test in a specific configuration (*e.g.*, on Windows, or using a specific version of the library, *etc.*), and in this case is difficult to avoid an if-else condition.

Eager Test: Eager Test is considered to have a low impact on test suite maintainability and is considered primarily as technical debt which can be addressed later or will not be addressed at all. Of the 51 reported cases, 26 times (50.98%) was reported as long-term refactoring, and 14 times was reported as no action/false positive. Only in 11 cases was considered as immediate refactoring.

Among the ”no action” feedback, a repetitive comment is that getters and setters should be left out from the analysis. In this case, by getters and setters we not only mean the Java encapsulation pattern (*i.e.*, getters and setters of class members), but all the methods that retrieve (*e.g.*, get) or set a variable in the production class. Listing 4.1 exemplifies a common won't fix Eager Test pattern.

```
@Test
public void testFooAction() {
    Foo foo = new Foo(...);
    foo.setProp1(...); //1st production method call
    foo.setProp2(...); //2nd call
    int result = foo.action(); //3rd call
    assertEquals(result, expectedResult);
    assertEquals(foo.getProp1(), ...); //4th call
    assertEquals(foo.getProp2(), ...); //5th call
}
```

Example 4.1: Example of a test case containing an Eager Test

Empty Test Empty test is considered as an immediate refactoring candidate and rated as having a high to very high impact on test suite maintainability. In this case, almost all the developers agreed that the test should be immediately removed, since it does not contain any assert statement.

General Fixture General Fixture is considered among the sampled developers as having a low to medium impact on the test suite maintainability and not considered as an issue to remove. In most of the cases, the tests used all or a different combination of the test fixture variables – on class level, all of the test fixtures were used.

Ignored Test Ignored Tests are considered to have a high impact on test suite maintainability and considered for long-term removal. Ignored tests are often accompanied by a bug ID inside the ignore statement (*e.g.*, `@Ignore("PROJECTCODE-123")`). Developers consider that these tests should not count as a test smell instance if they are properly

documented in the bug tracking system. The source of the bug might not be in the test but rather in the production code.

Magic Number Test Magic Number Test is considered low to medium severity and rated primarily as a short-term refactoring candidate. Tests exhibiting the Magic Number Test which were marked for refactoring were accompanied by comments from the developers, such as where the numbers come from and how they could be replaced with named constants (e.g., HTTP status codes). In instances marked as will not fix, the magic numbers were the results of a calculation performed inside the tested function. In these cases, the developers do not see the problem of not naming the constants, as the value according to them should be clear from the test context.

Mystery Guest Mystery Guest is rated as having medium to high impact on test suite maintainability and is considered to be primarily a candidate for immediate refactoring. Unless the developer wants to test the reading/writing of a file to the file system, a more appropriate approach would be to create a data structure that holds the necessary information.

Resource Optimism Resource Optimism is considered to be primarily a short-term refactoring candidate of high severity. The refactoring for this test smell can be done by adding file existence checks for each resource used.

Sleepy Test Sleepy Test is primarily perceived as having a high impact on test suite maintainability and is considered for immediate refactoring. The presence of thread sleep calls indicates that the test is not a good unit test and negatively impacts the test suite maintainability, since it might introduce flakiness. The removal of this test smell could require rewriting the entire test.

Verbose Test Verbose Test is split between immediate and long term refactoring. It is rated as having an overall medium to high impact on test suite maintainability. In the feedback, the developers pointed out that the tests exhibiting Verbose Test smell can be either split into multiple smaller tests or refactored to use methods to decrease the test size.

Analysis of 'will not fix' reports To gain more insights on why developers reported test smells as '`will not fix`', we examine all of the instances which had their source code accessible in a public repository (29 out of 60 ratings). We performed an open card sorting method [133] with two inspectors. This method allows creating mental models and allowing the definition of taxonomies from input data [133].

The two inspectors were the first two authors of this chapter, each with over four years of Java development experience. We applied the open card sorting method by first independently grouping the test smell instances and then comparing the results with a discussion about the classification and group differences. After we have agreed on a unified view on the classification, we move to the next test smell. The inspectors had at their disposal the source code of the test and the comments provided by the developer.

Based on the open card sorting methodology, we have identified the following four categories of will not fix test smells.

False Positive. These instances are true false positives – they show the limitations of our current tooling and approach. Here the fix would require modifying tsDetect or creating a new tool. The issue could not be fixed by setting better thresholds. There were a total of 7 false positives (24%).

Dismissed, Hard to fix. In this category, developers either do not see the problem with the test or changing the test to remove the smell instance would require a significant amount of effort to fix. The fix in some instances would require rewriting the production class under test, or even the related classes to support better testability. In these cases, the test smell points to larger problems of the codebase, not just related to the test design itself. There were a total of 11 instances (38%).

Dismissed, Easy Fix. Test smells in this category can be easily removed by refactoring them. However, developers either do not acknowledge them as a problem or are unwilling to refactor it to remove the test smell instance. In all identified cases in this category, the readability and maintainability of the test would be significantly improved if the test was refactored to follow the best practices. From the developer's point of view, these might be only perceived to be marginal gains and they concentrate their efforts elsewhere. There were 9 instances (31%) in this category.

Acknowledged problem, won't fix. Test smell instances in this category are acknowledged by the developers as possibly problematic, but in the domain context, they are perceived to be an acceptable trade-off between maintainability and ease of writing. The two instances of this category were Assertion Roulette smells. Here the developers consider "simple" assertions to be self-documenting. For example, in case of failing '`assertEquals( HttpStatus.BAD_REQUEST, ...);`' is easy to understand, even without documentation.

4.5 Discussion and Future Work

In this section we discuss our results, their implications, and future research directions.

Severity Rating of Test Smells. Our approach of defining test smell severity shows promising results when applied to certain test smells, as pointed out by the developers agreeing with our rating. On the other hand, for some test smells our approach does not work, as they are very rare, and an alternative classification would have to be used to define their severity.

We defined new severity thresholds for Assertion Roulette, Conditional Test Logic, Eager Test, and Verbose Test. These test smells had their respective metrics distributed across the selected benchmark project's codebase. On the contrary, several of our proposed thresholds had zero-valued thresholds up to or including the very high severity, which presents problems as we can not make any distinction between the severity levels. A possible approach to introduce granularity for the test smells where the thresholds are zeros would be to investigate the top 10 percentile of the worst projects, and extend the scale to there. Then set the high and very high thresholds based on the values of applying the original methodology on only this filtered sample size again. This would ensure that the medium risk threshold is at zero following the original methodology, and would introduce more fine-grained values for the high and very high thresholds.

Since out of the scope of this chapter, we treated the smells with zero thresholds as critical (very high) severity, because present in less than 10% of the benchmark corpus. However, in certain test smells such as Sleepy Test, the usage of even one thread sleep call versus multiple calls can be considered to be irrelevant, as even one such call makes the test dependent on the timing and can introduce multi-threading faults and flakiness.

Two test smells were excluded from the calibration: "Ignored Test" and "Empty Test". For these test smells, no conventional metric exists; hence an alternative should be proposed. For example, "Ignored Test" smell can be classified based on whether the @Ignore annotation contains a bug tracking issue identifier or description for the failing test. Developers can then focus on tests that are ignored without a description to investigate the root cause of the failure.

Classification of Empty Test could involve looking at whether the test is empty and does not contain any commented code. If it contains commented code, this could indicate that the test should have an `@Ignore` annotation instead, since it is masking an uncovered issue. In the latter case, these tests could be considered to be of high severity and the developers should investigate why the test code was commented out.

Developers' perception. By using the new derived thresholds we lowered the number of false positives the tool detects. However, as we saw from our RQ2, developers in some cases still do not perceive the smell as an actual problem and mark it as no-fix. The reason for this could be that the thresholds are still set too low, and most of the found instances are perceived as not problematic. This result is in line with previous studies [12], where the authors showed that often developers do not perceive test smells as actual problems. More recently, De Bleser *et al.* [130] studied developers' perception on test smells of Scala projects. The authors found that General Fixture and Mystery Guest are the test smells that are perceived as the most problematic. While we can confirm this result for Mystery Guest, the developers that participated in our study rated General Fixture as having a low impact on test suite maintainability.

Test Smell Triage. Test smells density, defined as the number of test smells in a test class, is too low level to be used as direct metrics to present to developers. It can be incorporated into another higher level metric to measure test code quality.

For example, a possible approach would be to look into test smell co-occurrence, combined with our proposed severity levels, to rank individual unit tests, test files, or test modules as ones that need more attention than the others. A test file impacted by several different instances of high severity test smells might need more attention than a file impacted by only one test smell type of low severity. This would give better actionability to the developers, as they would then have an indicator where to direct their attention. This approach would require investigating whether a combination of several test smells of different severity levels is worse than, for example, one very high severity test smell.

New or Improved Test Smell Detection Tool. We have identified several limitations in how tsDetect identifies test smells and that the current definition of certain test smells does not match the developer's perception or common practices. This mismatch between the developer's perception of what is vs. is not a test smell cannot be fixed by setting thresholds alone. In the case of Assertion Roulette, the developers consider good test naming to be sufficient to document the assertions. Incorporating or creating a test naming model, such as the one proposed by Meester [134] for rating how well a method is

named, could be used to determine if a test is named correctly and thus the assertions do not need an additional explanation. Currently, no tool also considers block comments above a series of assertions as a description, while developers consider this to be enough to describe what the assertions are for.

For Ignored Test, further research could be investigating whether it is possible to classify the description in the `@Ignore` annotation to determine if it contains a bug tracking issue identifier or a description for why it is failing. The impact of Eager Test detection excluding getters and setters could be investigated to see if this detection method would improve the probability that flagged Eager Tests are given bigger priority by the developers. All of the above could be further explored and then integrated into a new test smell detection tool, which would detect a smaller subset of test smells; however, the smells detected would be more likely to be perceived by the developers as problematic and an issue to remove.

4.6 Threats to Validity

Construct validity. Threats to construct validity concern our research instruments. We used the tool tsDetect to detect test smells and classify them according to our defined thresholds. For the subset of test smells we observed, the tool was reported to have an F-score surpassing 87% for each test smell as determined by a comprehensive review of multiple projects [117].

Internal validity. Threats to internal validity concern factors that could affect the variables and the relations being investigated. All of the contacted developers for this experiment were users of BCH, who have used it at least once in the past. The developers using a code quality tool might have more knowledge about code quality than developers who have never used such tools and thus their opinion might be different than those of developers who do not use BCH or any other quality control tool. Furthermore, developers that use code quality tools might care more of code quality (hence rating a test smell as high impact) than developers that do not use code quality tools.

External validity. Threats to external validity concern the generalization of results. We are aware that derivation of metrics thresholds from a benchmark dataset introduces dependency on the dataset and its representativeness. Further investigation needs to be done to determine the degree of sensitivity of this approach with respect to different benchmark datasets, precomputed metrics, and different metric extraction tools. However, to mitigate this issue, we used a big corpus of 1,489 OSS systems of different scopes, size and characteristics, to strengthen the generalizability of our findings.

For the second part of the study, we modified BCH by including a test smell detector and asked developers to rate their code. We have received responses from 31 developers, who evaluated 47 distinct projects. In comparison to other experiments in software engineering, our sample size is above the median (30 respondents) [135].

We only analyzed Java source code for test smells. The thresholds we defined would have to be defined per language, as the occurrence of test smells might be different across various languages and unit testing frameworks. The methodology for the test smell calibration should nonetheless generalize to other languages if they are supported by a test smell detection tool, which can also provide metrics in addition to detection.

4.7 Conclusion

In this chapter, we investigated the classification of test smells based on their severity. To do this, we define metrics for each test smell and then apply the benchmark-based threshold derivation on a sample of open-source projects. The result of this process allows us to give test smells a severity rating, from low to very high, which we verified with a sample of developers using their test source code to see if they agree with the rating classification. For four test smells (Assertion Roulette, Eager Test, Verbose Test and Conditional Test Logic), we defined non-binary severity thresholds and have a statistically significant difference between our proposed severity and user-perceived maintainability impact. Furthermore, Verbose Test and Conditional Test Logic showed a high statistically significant relationship and a strong Spearman coefficient.

We conducted a study of developer perception on selected test smells in their codebase using a modified version of BCH that allows for the detection of test smells with the new derived thresholds. We asked the developers to indicate for each test smell instance whether they would classify it as a valid instance to remove, how much priority they would give to the refactoring, and how long it would take according to them. Among the main results, we saw that Empty Test and Sleepy Test are considered the smells with the highest priority in refactoring, follows by Mystery Guest and Resource Optimism. The smells with the lowest priorities instead were General Fixture, Ignored Test, and Eager Test.

Furthermore, we analyzed test smells the developers marked as won't fix, trying to discover the reasons behind it, indicating how test smell detection tools could be improved to match the developer's views on what is a test smell. This information can also be used to enhance the proposed test smell severity ratings.

Furthermore, we have shown that test smell detection can be successfully integrated into a code quality tool to inform developers on test issues and help make tests more maintainable.

5

Mock Objects For Testing Java Systems: Why and How Developers Use Them, and How They Evolve

When testing software artifacts that have several dependencies, one has the possibility of either instantiating these dependencies or using mock objects to simulate the dependencies' expected behavior. Even though recent quantitative studies showed that mock objects are widely used both in open source and proprietary projects, scientific knowledge is still lacking on how and why practitioners use mocks. An empirical understanding of the situations where developers have (and have not) been applying mocks, as well as the impact of such decisions in terms of coupling and software evolution can be used to help practitioners adapt and improve their future usage. To this aim, we study the usage of mock objects in three OSS projects and one industrial system. More specifically, we manually analyze more than 2,000 mock usages. We then discuss our findings with developers from these systems, and identify practices and challenges, and support the results through a structured survey with more than 100 professionals. Finally, we manually analyze how the usage of mock objects in test code evolve over time as well as the impact of their usage on the coupling between test and production code.

Preprint: https://doi.org/10.1007/s10664-018-9663-0,

Data and materials: https://doi.org/10.5281/zenodo.4075346.

This chapter has been published as *Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. Mock objects for testing java systems: Why and how developers use them, and how they evolve. Empirical Software Engineering, 2019.* [136], an extension of

Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 402–412. IEEE, may 2017. [137]

5.1 Introduction

In software testing, it is common that the software artifact under test depends on other components [138]. Therefore, when testing a unit (*i.e.*, a class in object-oriented programming), developers often need to decide whether to test the unit and all its dependencies together (similar to an integration testing) or to *simulate* these dependencies and test the unit in isolation.

By testing all dependencies together, developers gain realism: The test will more likely reflect the behavior in production [139]. However, some dependencies, such as databases and web services, may (1) slow the execution of the test [7], (2) be costly to properly set up for testing [140], and (3) require testers to have full control over such external dependencies [39]. By simulating its dependencies, developers gain focus: The test will cover only the specific unit and the expected interactions with its dependencies; moreover, inefficiencies of testing dependencies are mitigated.

To support the simulation of dependencies, *mocking frameworks* have been developed (*e.g.*, Mockito [1], EasyMock [2], and JMock [3] for Java, Mock [4] and Mocker [5] for Python), which provide APIs for creating mock (*i.e.*, simulated) objects, setting return values of methods in the mock objects, and checking interactions between the component under test and the mock objects. Past research has reported that software projects are using mocking frameworks widely [141] [142] and has provided initial evidence that using a mock object can ease the process of unit testing [143].

Given the relevance of mocking, technical literature describes how mocks can be implemented in different languages [7, 38–42]. However, how and why practitioners use mocks, what kind of challenges developers face, and how mock objects evolve over time are still unanswered questions.

We see the answers to these questions as important to practitioners, tool makers, and researchers. Practitioners have been using mocks for a long time, and we observe that the topic has been dividing practitioners into two groups: The ones who support the usage of mocks (*e.g.*, Freeman and Pryce [39] defend the usage of mocks as a way to design how classes should collaborate among each other) and the ones who believe that mocks may do more harm than good (*e.g.*, as in the discussion between Fowler, Beck, and Hansson, well-known experts in the software engineering industry community [144, 145]). An empirical understanding of the situations where developers have been and have not been applying mocks, as well as the impact of such decisions in terms of coupling and software evolution, can be used to help practitioners adapt and improve their future usage. In addition, tool makers have been developing mocking frameworks for several languages. Although all these frameworks share the main goal, they take different decisions: As an example, JMock opts for strict mocks, whereas Mockito opts for lenient mocks.[6] Our findings can inform tool makers when taking decisions about which features practitioners really need

[1] http://site.mockito.org
[2] http://easymock.org
[3] http://www.jmock.org
[4] https://github.com/testing-cabal/mock
[5] https://labix.org/mocker
[6] When mocks are strict, the test fails if an unexpected interaction happens. In lenient mocks, tests do not fail for such reason. In Mockito 1.x, mocks are lenient by default; in Mockito 2.x, mocks are lenient, and by default, tests do not fail, and warnings happen when an unexpected interaction happens.

(and do not need) in practice. Finally, one of the challenges faced by researchers working on automated test generation concerns how to simulate a dependency [146, 147]. Some of the automated testing generation tools apply mock objects to external classes, but automatically deciding what classes to mock and what classes not to mock to maximize the test feedback is not trivial. Our study also provides empirical evidence on which classes developers mock, thus possibly indicating the automated test generation tools how to do a better job.

5.2 Background: Mock objects

"Once," said the Mock Turtle at last, with a deep sigh, "I was a real Turtle."
— Alice In Wonderland, Lewis Carroll

Mocking objects are a standard technique in software testing used to simulate dependencies. Software testers often mock to exercise the component under test in isolation.

Mock objects are available in most major programming languages. As examples, Mockito, EasyMock, as well as JMock are mocking frameworks available for Java, and Moq is available for C#. Although the APIs of these frameworks might be slightly different from each other, they provide developers with a set of similar functionalities: the creation of a mock, the set up of its behavior, and a set of assertions to make sure the mock behaves as expected. Listing 5.1 shows an example usage of Mockito, one of the most popular mocking libraries in Java [142]. We now explain each code block of the example:

1. At the beginning, one must define the class that should be mocked by Mockito. In our example, `LinkedList` is being mocked (line 2). The returned object (`mockedList`) is now a mock: It can respond to all existing methods in the `LinkedList` class.

2. As second step, we provide a new behavior to the newly instantiated mock. In the example, we inform the mock to return the string 'first' when the method `mockedList.get(0)` is invoked (line 5) and to throw a `RuntimeException` on `mockedList.get(1)` (line 7).

3. The mock is now ready to be used. In lines 10 and 11, the mock will answer method invocations with the values provided in step 2.

```
1  // 1: Mocking LinkedList
2  LinkedList mockObj = mock(LinkedList.class);
3
4  // 2: Instructing the mock object behaviour
5  when(mockObj.get(0)).thenReturn(" first ");
6  when(mockObj.get(1))
7  .thenThrow(new RuntimeException());
8
9  // 3: Invoking methods in the mock
10 System.out.println(mockObj.get(0));
11 System.out.println(mockObj.get(1));
```

Example 5.1: Example of an object being mocked

Typically, methods of mock objects are designed to have the same interface as the real dependency, so that the client code (the one that depends on the component we desire to mock) works with both the real dependency and the mock object. Thus, whenever developers do not want to rely on the real implementation of the dependency (*e.g.*, a database), they can simulate this implementation and define the expected behavior using the approach mentioned above.

5.2.1 Motivating example

Sonarqube is a popular open source system that provides continuous code inspection.[7] In January of 2017, Sonarqube contained over 5,500 classes, 700k lines of code, and 2,034 test units. Among all test units, 652 make use of mock objects, mocking a total of 1,411 unique dependencies.

Let us consider the class `IssueChangeDao` as an example. This class is responsible for accessing the database regarding changes in issues (changes and issues are business entities of the system). To that end, this class uses MyBatis [8], a Java library for accessing databases.

Four test units use `IssueChangeDao`. The dependency is mocked in two of them; in the other two, the test creates a concrete instance of the database (to access the database during the test execution). *Why do developers mock in some cases and do not mock in other cases?* Indeed, this is a key question motivating this work.

After manually analyzing these tests, we observed that:

- In Test 1, the class is concretely instantiated as this test unit performs an integration test with one of their web services. As the test exercises the web service, a database needs to be active.
- In Test 2, the class is also concretely instantiated as `IssueChangeDao` is the class under test.
- In both Test 3 and Test 4, test units focus on testing two different classes that use `IssueChangeDao` as part of their job.

This example reinforces the idea that deciding whether or not to mock a class is not trivial. Developers have different reasons which vary according to the context. In this work, we investigate patterns of how developers mock by analyzing the use of mocks in software systems and we examine their rationale by interviewing and surveying practitioners on their mocking practices. Moreover, we analyze data on how mocks are introduced and evolve.

5.3 Research Methodology

Our study has a twofold *goal*. First, we aim at understanding how and why developers apply mock objects in their test suites. Second, we aim at understanding how mock objects in a test suite are introduced and evolve over time.

[7] https://www.sonarqube.org/
[8] https://mybatis.org/mybatis-3/

Table 5.1: The studied sample in terms of size and number of tests (N=4).

Project	# of classes	LOC	# of test units	# of test units with mock
Sonarqube	5,771	701k	2,034	652
Spring Framework	6,561	997k	2,020	299
VRaptor	551	45k	126	80
Alura	1,009	75k	239	91
Total	13.892	1.818k	4.419	1.122

Table 5.2: The studied sample in terms of mock usage (N=4).

Project	# of mocked dependencies	# of not mocked dependencies	Sample size of mocked (CL=95%)	Sample size of not mocked (CL=95%)
Sonarqube	1,411	12,136	302	372
Spring Framework	670	21,098	244	377
VRaptor	258	1,075	155	283
Alura	229	1,436	143	302
Total	2,568	35,745	844	1,334

To achieve our first goal, we conduct quantitative and qualitative research focusing on four software systems and address the following questions:

RQ$_1$: What dependencies do developers mock in their tests? When writing an automated test for a given class, developers can either mock or use a concrete instance of its dependencies. Different authors [4, 148] affirm that mock objects can be used when a class depends upon some infrastructure (*e.g.*, file system, caching). We aim to identify what dependencies developers mock and how often they do it by means of manual analysis in source code from different systems.

RQ$_2$: Why do developers decide to (not) mock specific dependencies? We aim to find an explanation to the findings in previous RQ. We interview developers from the analyzed systems and ask for an explanation on why some dependencies are mocked while others are not. Furthermore, we survey software developers with the goal of challenging the findings from the interviews.

RQ$_3$: Which are the main challenges experienced with testing using mocks? Understanding challenges sheds light on important aspects on which researchers and practitioners can effectively focus next. Therefore, we investigate the main challenges developers face when using mocks by means of interviews and surveys.

To achieve our second goal, we analyze the mock usage history of the same four software systems and answer the following research questions:

RQ$_4$: When are mocks introduced in the test code? In this RQ, we analyze when mocks are introduced in the test class: Are they introduced together with the test

class, or are mocks part of the future evolution of the test? The answer to this question will shed light on how the behavior of software testers and their testing strategies when it comes to mocking.

RQ_5: How does a mock evolve over time? Practitioners affirm that mocks are highly coupled to the production class they mock [6]. In this RQ, we analyze what kind of changes mock objects encounter after their introduction in the test class. The answer to this question will help in understanding the coupling between mocks and the production class under test as well as their change-proneness.

5.3.1 Sample selection

We focus on projects that routinely use mock objects. We analyze projects that make use of Mockito, the most popular framework in Java with OSS projects [142].

We select three open source software projects (*i.e.*, Sonarqube,[9] Spring,[10] VRaptor[11]) and a software system from an industrial organization we previously collaborated with (Alura[12]). Tables 5.1 and 5.2 detail the size of these projects, as well as their mock usage. In the following, we describe their suitability to our investigation:

Spring Framework. Spring provides extensive infrastructural support for Java developers; its core serves as a base for many other offered services, such as dependency injection and transaction management. The Spring framework integrates with several other external software systems, which makes an ideal scenario for mocking.

Sonarqube. Sonarqube is a quality management platform that continuously measures the quality of source code and delivers reports to its developers. Sonarqube is a database-centric application, as its database plays an important role in the system.

VRaptor. VRaptor is an MVC framework that provides an easy way to integrate Java EE capabilities (such as CDI) and to develop REST web services. Similar to Spring MVC, the framework has to deal frequently with system and environment dependencies, which are good cases for mocking.

Alura. Alura is a proprietary web e-learning system used by thousands of students. It is a database-centric system developed in Java. The application resembles commercial software in the sense that it serves a single business purpose and makes heavy use of databases. According to their team leader, all developers make intensive use of mocking practices.

5.3.2 RQs 1, 2, 3: Data Collection and Analysis

The research method we use to answer our first three research questions follows a mixed qualitative and quantitative approach, which we depict in Figure 5.1: (**1**) We automatically collect all mocked and non-mocked dependencies in the test units of the analyzed systems, (**2**) we manually analyze a sample of these dependencies with the goal of understanding

[9] https://www.sonarqube.org/
[10] https://projects.spring.io/spring-framework/
[11] https://www.vraptor.com.br/
[12] http://www.alura.com.br/

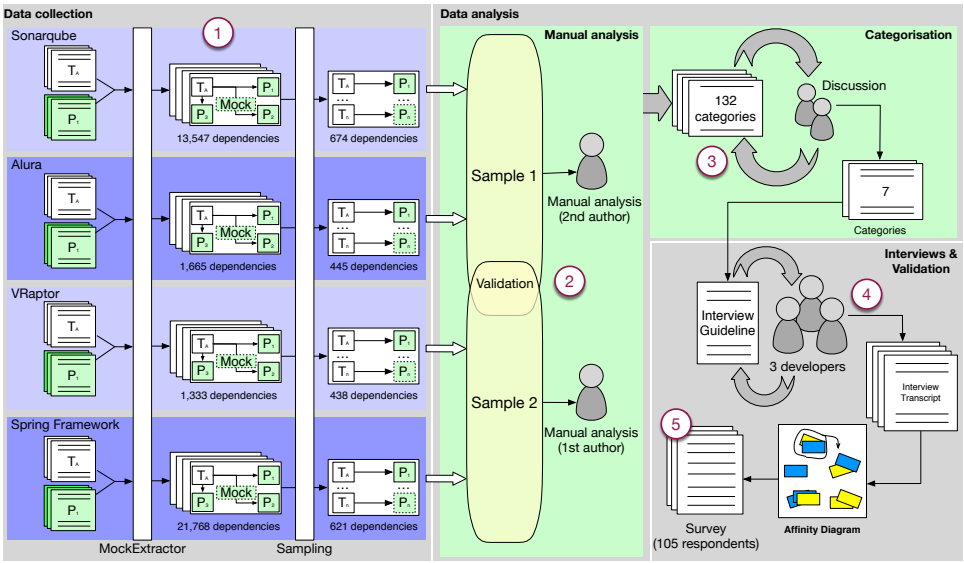


Figure 5.1: The mixed approach research method applied.

their architectural concerns as well as their implementation, (**3**) we group these architectural concerns into categories, which enables us to compare mocked and non-mocked dependencies among these categories, (**4**) we interview developers from the studied systems to understand our findings, and (**5**) we enhance our results in an online survey with 105 respondents.

1. Data collection. To obtain data on mocking practices, we first collect all the dependencies in the test units of our systems performing static analysis on their test code. To this aim, we create MockExtractor [149], a tool that implements the algorithm below:

1. We detect all test classes in the software system. As done in past literature (*e.g.*, Zaidman *et al.* [56]), we consider a class to be a test when its name ends with 'Test' or 'Tests.'

2. For each test class, we extract the (possibly extensive) list of all its dependencies. Examples of dependencies are the class under test itself, its required dependencies, and utility classes (*e.g.*, lists and test helpers).

3. We mark each dependency as 'mocked' or 'not mocked.' Mockito provides two APIs for creating a mock from a given class:[13] (1) By making use of the `@Mock` annotation in a class field or (2) by invoking `Mockito.mock()` inside the test method. Every time one of the two options is found in the code, we identify the type of the class that is mocked. The class is then marked as 'mocked' in that test unit. If a dependency

[13]Mockito can also generate *spies* which are out of the scope of this chapter. More information can be found in Mockito's documentation: http://bit.ly/2kjtEi6.

appears more than once in the test unit, we consider it 'mocked.' A dependency may be considered 'mocked' in one test unit, but 'not mocked' in another.

4. We mark dependencies as 'not mocked' by subtracting the mocked dependencies from the set of all dependencies.

2. Manual analysis. To answer what test dependencies developers mock, we analyze the previously extracted mocked and non-mocked dependencies. The goal of the analysis is to understand the main concern of the class in the architecture of the software system (*e.g.*, a class is responsible for representing a business entity, or a class is responsible for persisting into the database). Defining the architectural concern of a class is not an easy task to be automated, since it is context-specific, thus we decided to perform a manual analysis. The first two authors of the chapter conducted this analysis after having studied the architecture of the four systems.

Due to the size of the total number of mocked and non-mocked dependencies (around 38,000), we analyze a random sample. The sample is created with the confidence level of 95% and the error (E) of 5%, *i.e.*, if in the sample a specific dependency is mocked $f\%$ of the times, we are 95% confident that it will be mocked $f\% \pm 5\%$ in the entire test suite. Since projects belong to different areas and results can be completely different from each other, we create a sample for each project. We produce four samples, one belonging to each project. This gives us fine-grained information to investigate mock practices within each project.

In Table 5.1 we show the final number of analyzed dependencies ($844 + 1,334 = 2,178$ dependencies).

The manual analysis procedure is as follows:

- Each researcher is in charge of two projects. The selection is made by convenience: The second author focuses on VRaptor and Alura, since he is already familiar with their internal structure.
- All dependencies in the sample are listed in a spreadsheet to which both researchers have access. Each row contains information about the test unit where the dependency was found, the name of the dependency, and a boolean indicating if that dependency was mocked.
- For each dependency in the sample, the researcher manually inspects the source code of the class. To fully understand the class' architectural concern, researchers can navigate through any other relevant piece of code.
- After understanding the concern of that class, the researcher fills the "Category" column with what best describes the concern. No categories are defined up-front. In case of doubt, the researcher first reads the test unit code; if not enough, he then talks with the other research.
- At the end of each day, the researchers discuss together their main findings and some specific cases.

The entire process took seven full days. The total number of categories was 116. We then start the second phase of the manual analysis, focused on *merging categories*.

Table 5.3: Profile of the interviewees

Project	ID	Role in the project	Years of programming experience
Spring Framework	D1	Lead Developer	25
VRaptor	D2	Lead Developer	10
Alura	D3	Lead Developer	5

3. Categorization. To group similar categories we use a technique similar to card sorting [150]: (**1**) each category is represented in a card, (**2**) the first two authors analyze the cards applying open (*i.e.*, without predefined groups) card sort, (**3**) the researcher who created the category explain the reasons behind it and discuss a possible generalization (to make the discussion more concrete it is allowed to show the source code of the class), (**4**) similar categories are then grouped into a final, higher level category. (**5**) at the end, the authors give a name to each *final* category.

After following this procedure for all the 116 categories, we obtained a total of 7 categories that describe the concerns of classes.

The large difference between 116 and 7 is the result of most concerns being grouped into two categories: 'Domain object' and 'External dependencies.' The former classes always represent some business logic of the system and has no external dependencies. The full list of the 116 categories is available in our on-line appendix [48].

4. Interviews. We use the results from our investigation on the dependencies that developers mock (RQ_1) as an input to the data collection procedure of RQ_2. We design an interview in which the goal is to understand *why* developers did mock some roles and did not mock other roles. The interview is semi-structured and is conducted by the first two authors of this chapter. For each finding in previous RQ, we ensure that the interviewee describes why they did or did not mock that particular category, what the perceived advantages and disadvantages are, and any exceptions to this rule. Our full interview protocol is available in the appendix [48].

As a selection criterion for the interviews, we aimed at technical leaders of the project. Our conjecture was that technical leaders are aware of the testing decisions that are taken by the majority of the developers in the project. In practice, this turned out to be true, as our interviewees were knowledgeable about these decisions and talked about how our questions were also discussed by different members of their teams.

To find the technical leaders, we took a different approach for each project: in Alura (the industry project), we asked the company to point us to their technical leader. For VRaptor and Spring, we leveraged our contacts in the community (both developers have participated in previous research conducted by our group). Finally, for Sonarqube, as we did not have direct contact with developers, we emailed the top 15 contributors of the projects. Out of the 15, we received only a single (negative) response.

At the end, we conduct three interviews with active, prolific developers from three projects. Table 5.3 shows the interviewees' details.

We start each interview by asking general questions about interviewees' decisions with respect to mocking practices. As our goal is to explain the results we found in the

previous RQ (the types of classes, e.g., database and domain objects, as well as how often each of them is mocked by developers), we present the interviewee with two tables: one containing the numbers of each of the six categories in the four analyzed projects (see RQ_1 results, Figure 5.3), and another containing only the results of the interviewee's project.

We do not show specific classes, as we conjecture that remembering a specific decision in a specific class can be harder to remember than the general policy (or the "rule of thumb") that they apply for certain classes. Throughout the interview, we reinforce that participants should talk about the mocking decisions in their specific project (which we are investigating); divergent personal opinions are encouraged, but we require participants to explicitly separate them from what is done in the project. To make sure this happens, as interviewers, we question participants whenever we notice an answer that did not precisely match the results of the previous RQ.

As aforementioned, for each category, we present the findings and solicit an interpretation (*e.g.*, by explaining why it happens in their specific project and by comparing with what we saw in other projects). From a high-level perspective, we ask:

1. Can you explain this difference? Please, think about your experience with this project in particular.
2. We observe that your numbers are different when compared to other projects. In your opinion, why does it happen?
3. In your experience, when should one mock a <category>? Why?
4. In your experience, when should one not mock a <category>? Why?
5. Are there exceptions?
6. Do you know if your rules are also followed by the other developers in your project?

Throughout the interview, one of the researchers is in charge of summarizing the answers. Before finalizing the interview, we revisit the answers with the interviewee to validate our interpretation of their opinions. Finally, we close the interview by asking questions about the current challenges they face when applying mock practices in their projects.

Interviews are conducted via Skype and fully recorded, as well as manually transcribed by the researchers. With the full transcriptions, we perform card sorting [151, 152] to identify the main themes.

As a complement to the research question, whenever feasible, we also validate interviewees' perceptions by measuring them in their own software system.

5. Survey. To challenge and expand the concepts that emerge during the previous phases, we conduct a survey. All questions are derived from the results of previous RQs. The survey has four main parts. (1) In the first part, we ask respondents about their experience in software development and mocking. (2) The second part of the survey asks respondents about how often they make use of mock objects in each of the categories found during the manual analysis. (3) The third part asks respondents about how often

they mock classes in specific situations, such as when the class is too complex or coupled. (4) The fourth part focuses on challenges with mocking. Except for the last question, which is open-ended and optional, the questions are closed-ended and based on a 5-point Likert scale.

We initially design the survey in English, then we compile a Brazilian Portuguese translation, to reach a broader, more diverse population. Before deploying the survey, we first performed a pilot of both versions with four participants; we improved our survey based on their feedbacks (changes were all related to phrasing). We then shared our survey via Twitter (authors tweeted in their respective accounts), among our contacts, and in developers' mailing lists. The survey ran for one week. We analyze the open questions by performing card sorting. The full survey can be found in our on-line appendix [48].

We received a total of 105 answers from both Brazilian Portuguese and English surveys. The demographics of the participants can be found in Figure 5.2. 22% of the respondents have between one and five years of experience, 64% between 6 and 15 and 14% have more than 15 years of experience. The most used programming languages are Java (52%), JavaScript (42%), and C# (39%). Among the respondents, the most used mocking framework is Mockito (53%) followed by Moq (31%) and Powermock (8%). Furthermore, 66% of the participants were from South America, 21% from Europe, 8% from North America, and the remaining 5% from India and Africa. Overall our survey reached developers from different experience levels, programming languages, and mocking framework.

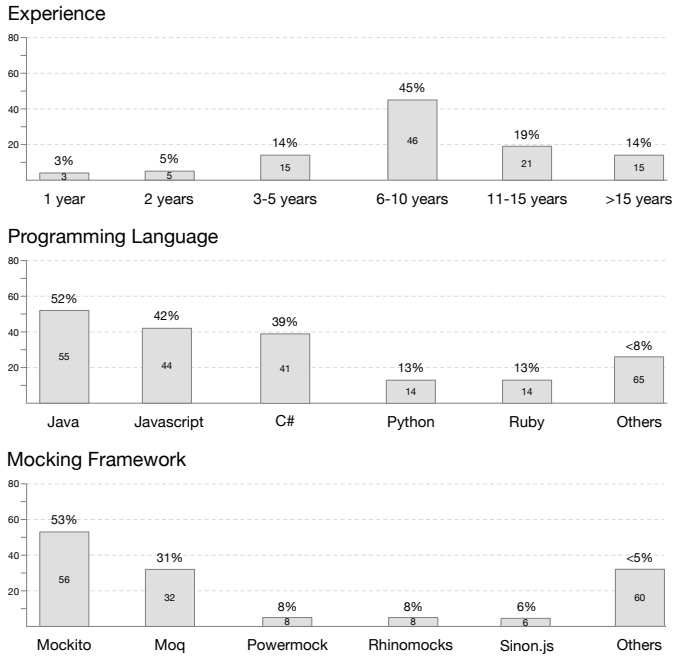


Figure 5.2: High-level survey participant details (n=105). Figures on Programming language and Mocking framework are higher than 100%, as participants could choose multiple options.

Table 5.4: The Mockito APIs we study the evolution in the four software systems.

Mockito API	Provided functionality
verify()	Verifies certain behavior happened once.
when()	Enables stubbing methods.
mock()	Creates a mock with some non-standard settings.
given()	Syntax sugar for BDD [153] practitioners. Same as when().
doThrow()	Stub the void method with an exception.
doAnswer()	Stub a void method with generic Answer.
doNothing()	Setting void methods to do nothing.
doCallRealMethod()	Call the real implementation of a method.
doReturn()	Similar to when(). Use only in the rare occasions when you cannot use when().
verifyZeroInteractions()	Verifies that no interactions happened on given mocks beyond the previously verified interactions.
verifyNoMoreInteractions()	Checks if any of given mocks has any unverified interaction.
thenReturn()	Sets a return value to be returned when the method is called.
thenThrow()	Sets a Throwable type to be thrown when the method is called.

5.3.3 RQ_4 and RQ_5: Data Collection and Analysis

To understand how mock objects evolve over time and what type of changes developers perform on them, we (i) collect information about test classes that make use of mocks and (ii) manually analyze a sample to understand how and why mocking code changes.

1. Data extraction We extract information about (1) when mock objects are introduced in test classes and (2) how the mocking code changes over time. To this aim, we create a static analysis tool and mine the history of the four analyzed systems. The tool implements the algorithm below:

For the mock introduction:

1. For each class in a commit, we identify all test classes. As done for MockExtractor, we consider a class to be a test when its name ends with `Test` or `Tests`. For each test class, we check if it makes use of at least one mock object, by checking whether the class imports Mockito dependencies.
2. Given a test class and an indication of whether that test class makes use of mocks, we classify the change as follows:
   (a) If the test class contains mocks and the test class is new (*i.e.*, Git classifies this modification as an addition), we consider that mocks were *introduced from its creation.*

(b) If the test class does not contain mocks, but it previously did, we consider that *mocks were removed* from that test class.

(c) If the test class contains mocks and it is a modification of a test:

i. If the test did not contain mocks before this change, we consider that mocks were *introduced* later in the test class' lifespan.

ii. If the test contained mocks before this change, we keep the current information we have about this class.

For the mock evolution:

3. For each test class that uses at least one mock object, we check whether developers modify any code line involving a mock. Since Mockito provides many APIs, we keep track only of the most used ones [142]. The APIs that we considered are depicted in Table 5.4. First, we obtain the list of modified lines of each test that involve Mockito APIs. Then, for each line:

   (a) We check the type of change, namely addition, deletion, or modification. To capture the latter, we adopt a technique similar to the one proposed by Biegel et al. [88], based on the use of textual analysis. Specifically, if the cosine similarity [89] between two lines in the diff of the previous version of the file and the new version of the file is higher than α, then we consider the two lines as a modification of the same line.

   (b) We obtain the list of Mockito APIs involved in the line using regular expressions.

   (c) Depending on the type of action we discover before, we update the number of times the corresponding Mockito API was added, deleted or changed.

   To determine the α threshold, we randomly sample a total of 75 real changes from the four software systems and test the precision of different thresholds, from 1% to 100%. Based on this process, we choose α = 0.71 (which leads to a precision of 73%).

2. Manual analysis. The goal of the analysis is to understand what drives mock objects to change once they are in a test class. To that aim, we manually analyze changes in the mocks extracted in the previous step. We opt for manual analysis as it is necessary to understand the context of the change.

Due to the size of the total number of modified lines involving Mockito APIs (~10,000), we analyzed a random sample. Similarly to our previous manually analysis, to obtain more fine-grained information, we create a sample of 100 changes for each project. This sample gives us a confidence level of 95% and an error (E) of 10%.

The manual analysis procedure is as follows:

- All the mock changes in the sample are listed in a spreadsheet. Each row contains information about the commit that modifies the line (*i.e.*, the commit hash), and how the line changed (its previous and successive versions).

- For each mock that changed, researchers manually inspect the change, with the goal of understanding the reasoning behind the change, *i.e.*, why the mock changed. To fully understand why the change happened, researchers also inspect the respective commit and the changes involved in there. This step gives information not only about the change in the mock itself, but also about the possible changes in production class being mocked, the test class, and the class under test. The researchers are not aware of the details of the project, and thus, they are not able to explain the change from the business perspective of the project (e.g., a mock changed due to a new feature that is introduced); rather, researchers focus on understanding whether the change was caused either because the test changed or because the production code changed and forced the mock to change together.
- After understanding all the context of the change and the reason behind the mock being changed, the researcher attributes a code (*i.e.*, reason) that best describes the change. As done for the first part of our study, no categories are defined up-front.
- The first 20 changes are done together by the two researchers so that both could adapt to the process. After, each researcher is in charge of two projects. During the entire analysis, researchers discuss odd cases as well as share and iteratively refine their code book.

5.4 Results

In this section, we present the results to our research questions aimed at understanding how and why developers apply mock objects in their test suites, the challenges developers face in this context, as well as the introduction and evolution of mocks.

RQ_1: What dependencies do developers mock in their tests?

As visible in Table 5.1, we analyzed 4,419 test units of which 1,122 (25.39%) contain at least one mock object. From the 38,313 collected dependencies from all test units, 35,745 (93.29%) are not mocked while 2,568 (6.71%) are mocked.

Since the same dependency may appear more than once in our dataset (*i.e.*, a class can appear in multiple test units), we calculated the *unique* dependencies. We obtain a total of 11,824 non-mocked and 938 mocked dependencies. The intersection of these two sets reveals that 650 dependencies (70% of all dependencies mocked at least once) were both mocked and non-mocked in the test suite.

In Figure 5.3, we show how often each role is mocked in our sample in each of the seven categories found during our manual analysis. One may note that 'databases' and 'web services' can also fit in the 'external dependency' category; we separate these two categories as they appear more frequently than other types of external dependencies. In the following, we detail each category:

Domain object: Classes that contain the (business) rules of the system. Most of these classes usually depend on other domain objects. They do not depend on any external resources. The definition of this category fits well with the definition of Domain Object [154] and Domain Logic [155] architectural layers. Examples are entities, services, and utility classes.

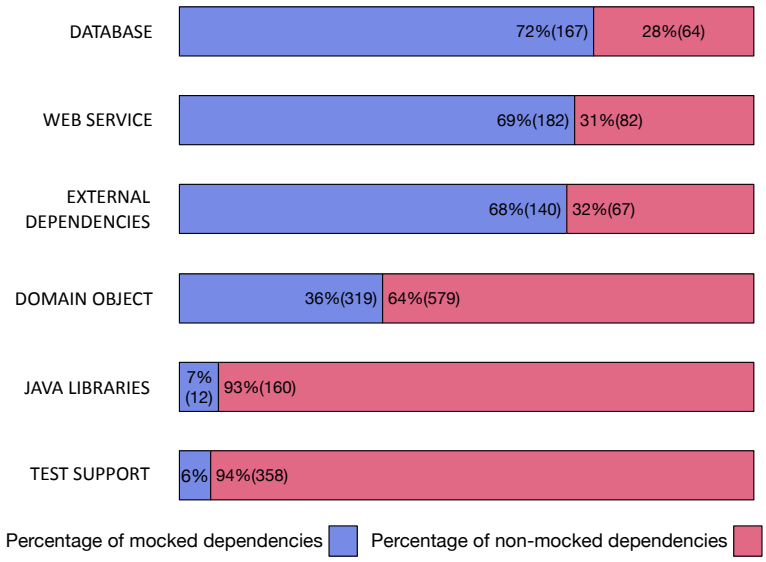


Figure 5.3: How often each architectural role is (not) mocked in the analyzed systems (N = 2,178)

Database: Classes that interact with an external database. These classes can be either an external library (such as Java SQL, JDBC, Hibernate, or ElasticSearch APIs) or a class that depends on such external libraries (*e.g.*, an implementation of the Data Access Object [155] pattern).

Native Java libraries: Libraries that are part of the Java itself. Examples are classes from Java I/O and Java Util classes (Date, Calendar).

Web Service: Classes that perform some HTTP action. As with the database category, this dependency can be either an external library (such as Java HTTP) or a class that depends on such library.

External dependency: Libraries (or classes that make use of libraries) that are external to the current project. Examples are Jetty and Ruby runtimes, JSON parsing libraries (such as GSON), e-mail libraries, etc.

Test support: Classes that support testing itself. Examples are fake domain objects, test data builders and web services for tests.

Unresolved: Dependencies that we were not able to solve. For example, classes belonging to a sub-module of the project for which the source code is not available.

Numbers are quite similar when we look at each project separately. Exceptions are for databases (Alura and Sonarqube mock ~60% of databases dependencies, Spring mocks 94%) and domain objects (while other projects mock them ~30% of times, Sonarqube mocks 47%).[14]

[14]We present the numbers for each project in our online appendix [48].

We observe that 'Web Services' and 'Databases' are the most mocked dependencies. On the other hand, there is no clear trend in 'Domain objects': numbers show that 36% of them are mocked. Even though the findings are aligned with the technical literature [4, 156], further investigation is necessary to understand the real rationale behind the results.

In contrast 'Test support' and 'Java libraries' are almost never mocked. The former is unsurprising since the category includes fake classes or classes that are created to support the test itself.

> ***RQ*$_1$.** *Classes that deal with external resources, such as databases and web services, are often mocked. There is no clear trend for domain objects.*

RQ$_2$. Why do developers decide to (not) mock specific dependencies?

In this section, we summarize the answers obtained during our interviews and surveys. We refer to the interviewees by their ID in Table 5.3.

Mocks are used when the concrete implementation is not simple. All interviewees agree that certain dependencies are easier to mock than to use their concrete implementation. They mentioned that classes that are highly coupled, complex to set up, contain complex code, perform a slow task, or depend on external resources (*e.g.*, databases, web services or external libraries) are candidates to be mocked. D2 gives a concrete example: *"It is simpler to set up an in-memory list with elements than inserting data into the database."* Interviewees affirmed that whenever they can completely control the input and output of a class, they prefer to instantiate the concrete implementation of the class rather than mocking it. As D1 stated: *"if given an input* [the production class] *will always return a single output, we do not mock it."*

In Figure 5.4, we see that survey respondents also often mock dependencies with such characteristics: 48% of respondents said they always or almost always mock classes that are highly coupled, and 45.5% when the class difficult to set up. Contrarily to our interviewees, survey respondents report to mock less often when it comes to slow or complex classes (50.4% and 34.5% of respondents affirm to never or almost never mock in such situations, respectively).

Mocks are not used when the focus of the test is the integration. Interviewees explained that they do not use mocks when they want to test the integration with an external dependency itself, (*e.g.*, a class that integrates with a database). In these cases they prefer to perform a real interaction between the unit under test and the external dependency. D1 said *"if we mock* [the integration], *then we wouldn't know if it actually works.* [...] *I do not mock when I want to test the database itself; I want to make sure that my SQL works. Other than that, we mock."* This is also confirmed in our survey (Figure 5.4), as our respondents also almost never mock the class under test.

The opposite scenario is when developers want to *unit* test a class that depends on a class that deals with external resources, (*e.g.*, `Foo` depends on `Boo`, and `Boo` interacts with a database). In this case, developers want to test a single unit without the influence of the external dependencies, thus developers evaluate whether they should mock that

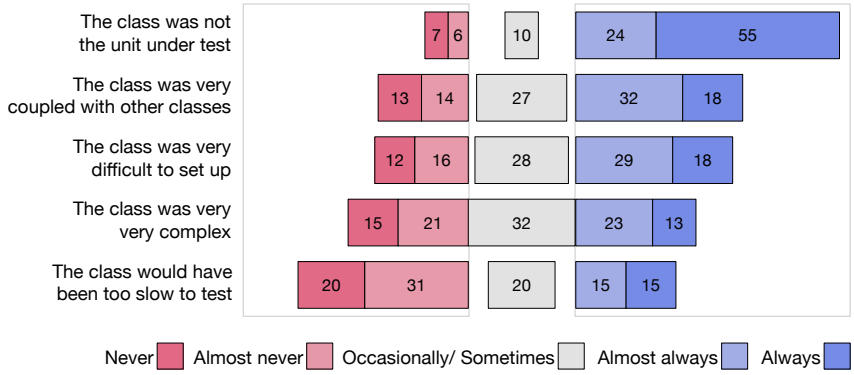


Figure 5.4: Reasons to use mock objects (N = 105)

dependency. D2 said: *"in unit testing, when the unit I wanna test uses classes that integrate with the external environment, we do not want to test if the integration works, but if our current unit works,* [...] *so we mock the dependencies."*

Interfaces are mocked rather than specific implementations. Interviewees agree that they often mock interfaces. They explain that an interface can have several implementations and they prefer to use a mock to not rely on a specific one. D1 said: *"when I test operations with side effects* [sending an email, making an HTTP Request] *I create an interface that represents the side effect and* [instead of using a specific implementation] *I mock the interface directly."*

Domain objects are usually not mocked. According to the interviewees, domain objects are often plain old Java objects, commonly composed by a set of attributes, getters, and setters. These classes also commonly do not deal with external resources; thus, these classes tend to be easily instantiated and set up. However, if a domain object is complex (*i.e.*, contains complicated business logic or not easy to set up), developers may mock them. Interviewee D2 says: "[if class `A` depends on the domain object `B`] *I'd probably have a* `BTest` *testing* `B` *so this is a green light for me to know that I don't need to test* `B` *again."* All interviewees also mention that the same rule applies if the domain object is highly coupled.

Figure 5.5 shows that answers about mocking 'Domain objects' vary. There is a slight trend towards not mocking them, in line to our findings during the interviews and in RQ_1.

Native Java objects and libraries are usually not mocked. According to D1, native Java objects are data holders (*e.g.*, `String` and `List`) that are easy to instantiate with the desired value. Thus no need for mocking. D1 points out that some native classes cannot even be mocked as they can be final (*e.g.*, `String`). D2 discussed the question from a different perspective. According to him, developers can trust the provided libraries, even

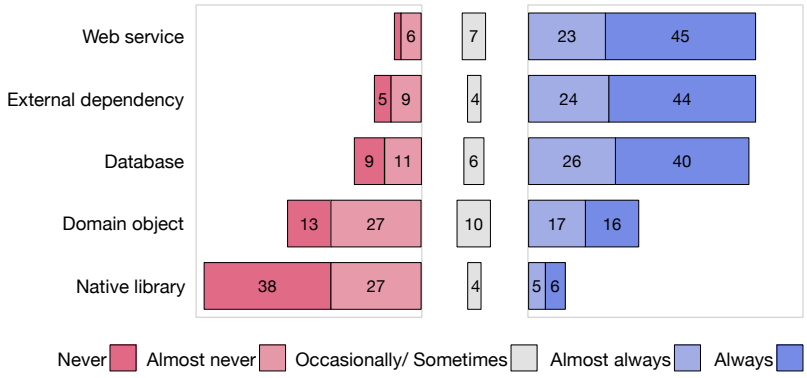


Figure 5.5: Frequency of mocking objects per category (N = 105)

though they are "external," thus, there is no need for mocking. Both D1 and D2 made an exception for the Java I/O library: According to them, dealing with files can also be complex, and thus, they prefer to mock. D3, on the other hand, affirms that in their software, they commonly do not mock I/O as they favor integration testing.

These findings match our data from RQ1, where we see that 'Native Java Libraries' are almost never mocked. Respondents also had a similar perception: 82% of them affirm to never or almost never mock such dependencies.

Database, web services, and external dependencies are slow, complex to set up, and are good candidates to be mocked. According to the interviewees, that is why mocks should be applied in such dependencies. D2 said: *"Our database integration tests take 40 minutes to execute, it is too much"*. These reasons also match with technical literature [4, 156].

All participants have a similar opinion when it comes to other kinds of external dependencies/libraries, such as CDI or a serialization library: When the focus of the testing is the integration itself, they do not mock. Otherwise, they mock. D2 said: *"When using CDI* [Java's Contexts and Dependency Injection API]*, it is really hard to create a concrete* [CDI] *event: in this case, we usually prefer to mock it"*. Two interviewees (D1 and D2) affirmed that libraries commonly have extensive test suites, thus developers do not need to "re-test". D3 had a different opinion: Developers should re-test the library as they cannot always be trusted.

In Figure 5.5, we observe that respondents always or almost always mock 'Web services' (~82%), 'External dependencies' (~79%) and 'Databases' (~71%). This result confirms the previous discovery that when developers do not want to test the integration itself, they prefer to mock these dependencies.

RQ$_2$. *The architectural role of the class is not the only factor developers take into account when mocking. Respondents report to mock when using the concrete implementation would be problematic (*e.g.*, the class would be too slow or complex to set up).*

RQ3. Which are the main challenges experienced with testing using mocks?

We summarize the main challenges that appeared in the interviews and in the answers of our survey question about challenges (which received 61 responses). Categories below represent the main themes that emerged during card sorting.

Dealing with coupling. Mocking practices deal with different coupling issues. On one hand, the usage of mocks in test increases the coupling between the test and the production code. On the other hand, the coupling among production classes themselves can also be challenging for mocking. According to a participant, *"if the code has not been written with proper decoupling and dependency isolation, then mocking is difficult (if not impossible)."* This matches with another participant's opinions who mentions to not have challenges anymore, by having *"learned how to separate concepts."*

Mocking in legacy systems. Legacy systems can pose some challenges for users of mocks. According to a respondent, testing a single unit in such systems may require too much mocking (*"to mock almost the entire system"*). Another participant even mentions the need of using *PowerMock* [15] (a framework that enables Java developers to mock certain classes that might be not possible without bytecode manipulation, *e.g.*, final classes and static methods) in cases where the class under test is not designed for testability. On the other hand, mocking may be the only way to perform unit testing in such systems. According to a participant: *"in legacy systems, where the architecture is not well-decoupled, mocking is the only way to perform some testing."*

Non-testable/Hard-to-test classes. Some technical details may impede the usage of mock objects. Besides the lack of design by testability, participants provide different examples of implementation details that can interfere with mocking. Respondents mentioned the use of static methods in Java (which are not mockable by default), file uploads in PHP, interfaces in dynamic languages, and the LINQ language feature in C#.

The relationship between mocks and good quality code. Mocks may reduce test readability and be difficult to maintain. Survey respondents state that the excessive use of mocks is an indicator of poorly engineered code. During the interviews, D1, D2, and D3 mentioned *the same example* where using mocks can hide a deeper problem in the system's design: indeed, they all said that a developer could mock a class with a lot of dependencies (to ease testing), but the problem would remain, since a class with a lot of dependencies probably represents a design flaw in the code. In this scenario, they find it much easier to mock the dependency as it is highly coupled and complex. However, they say this is a symptom of a poorly designed class. D3 added: *"good* [production] *code ease the process of testing. If the* [production] *code structure is well defined, we should use less*

[15] https://github.com/powermock/powermock

mocks". Interviewee D3 also said *"I always try to use as less mocks as possible, since in my opinion they hide the real problem. Furthermore, I do not remember a single case in which I found a bug using mocks'.* A survey respondent also shares the point that the use of mocks does not guarantee that your code will behave as expected in production: *"You are always guessing that what you mock will work (and keep working) that way when using the real objects."*

Unstable dependencies. A problem when using mocks is maintaining the behavior of the mock compatible with the behavior of the original class, especially when the original class is poorly designed or highly coupled. As the production class tends to change often, the mock object becomes unstable and, as a consequence, more prone to change.

> ***RQ*$_3$*.*** *The use of mocks poses several challenges, such as maintaining the behavior of the mock compatible with the original class, the relationship between how much mocking is necessary is to test a class and its code quality, and the (not positive) excessive use of mock objects to test legacy systems.*

RQ$_4$. Why and how do developers review test code?

In Table 5.5, we summarize the information about the introduction of mock objects in the four studied systems. We observe that:

- The vast majority of mocks (2,159, or 83% of the cases) are introduced at the inception of the test class. In a minority of cases (433, or 17% of the cases), mocks are introduced later in the lifetime of the test class. These results are consistent across the four studied systems, with an approximate 80/20 ratio. These results seem to indicate that developers generally tend to *not* refactor test cases to either introduce mocks or to delete them, instead they use mocks mostly for new tests. We hypothesize that this behavior may hint at the fact that developers (i) do not consider important to refactor their test code (as found in previous research [56], although this behavior could lead to critical test smells [2]), (ii) do not consider mocks a good way to improve the quality of existing test code, or (iii) start with a clear mind on whether they should use mocks to test a class. Investigating these hypotheses goes beyond the scope of this work, but studies can be devised to understand the reasons why developers adopt this behavior.

- Of these mocks, 343 (13%) were removed afterward (we consider both the cases in which the mock is introduced from the inception of the test class and the cases when the mock is introduced in a later moment). To have a more clear idea of what these removals correspond to, we manually inspect 20 cases. We observe that, in most cases, developers replace the mock object with the real implementation of the class. Despite this manual analysis, since we are not developers of the system, it is hard to pinpoint the underlying reason for the changes that convert tests from using mocks to using the real implementation of a class. Nevertheless, it is reasonable to hypothesize that the choice of deleting a mock is influenced by many different

factors, as it happens for the choices of (not) mocking a class, which we reported in the previous sections.

Table 5.5: When mock objects were introduced (N=2,935).

	Spring	Sonarqube	VRaptor	Alura	Total
Mocks introduced from the beginning	234 (86%)	1,485 (84%)	177 (94%)	263 (74%)	2,159 (83%)
Mocks introduced later	37 (14%)	293 (16%)	12 (6%)	91 (26%)	433 (17%)
Mocks removed from the tests	59 (22%)	243 (14%)	6 (3%)	35 (10%)	343 (13%)

RQ$_4$. *In the studied systems, mocks are mostly (80% of the time) present at the inception of the test class and tend to stay in the test class for its whole lifetime (87% of the time).*

RQ$_5$. Does availability bias affect the code review outcome?

Table 5.6 shows the evolution of the Mockito APIs over time. For each API, we have three categories: 'added,' 'changed,' and 'removed.' These categories correspond to the situations in which the lines containing the calls to the APIs are modified; we map the categorization of added/changed/removed as it provided by git. Furthermore, Table 5.6 reports the statistics for each project and a summary in the last column.

As expected[16] the API calls `verify()`, `when()`, `mock()` and `thenReturn()` are the most used ones. These results are complementing previous studies that reported similar findings [142]. Furthermore, we note that in Sonarqube developers make intense use of mock objects, especially by using the `when()` and `thenReturn()` APIs. Spring Framework is the only project that uses the `given()` API, which belongs to BDD Mockito, a set of APIs for writing tests according to the Behavior Driven Development process [153].

The most deleted APIs are `verify()`, `when()`, `mock()`, and `thenReturn()`. This is expected since these are the most frequently used APIs, thus it is more likely that they are later deleted. However, as to why they are deleted, reasons may be many. We see two plausible explanations, also taking into account the results from RQ$_4$: The first explanation for deletions is that the developer decided to delete the test, maybe because obsolete or not useful anymore; the second explanation is instead that the developer decided to replace the mock object with the real implementation of the production class.

Turning our attention to the Mockito APIs that change the most ('Changed calls to API' in Table 5.6), Table 5.7 shows the results of our manual analysis on a sample of more than 300 of these changes. We found that there are mainly two reasons for mocks to change: (a) the production code induced it (113 + 59 = 172, or 51%) or (b) improvements to the test code triggered it (164, or 49%). We detail these two cases in the following:

[16]In fact, it is not possible to correctly use Mockito without these API calls.

Table 5.6: The evolution of the Mockito APIs over time (N=74,983).

Mockito API	Spring framework	Sonarqube	VRaptor	Alura	Total
Added calls to API					
verify()	3,767	5,768	685	857	11,077
when()	533	9,531	1,771	1,961	13,796
mock()	1,316	6,293	334	590	8,533
given()	1,324	0	0	9	1,333
doThrow()	42	139	57	5	243
doAnswer()	1	13	1	0	15
doNothing()	11	15	3	0	29
doReturn()	5	111	19	24	159
verifyZeroInteractions()	22	556	15	35	628
verifyNoMoreInteractions()	115	605	3	6	729
thenReturn()	454	8,602	1,671	1,850	12,577
thenThrow()	8	135	30	12	185
Changed calls to API					
verify()	316	2,253	53	207	2,829
when()	53	2,731	134	386	3,304
mock()	121	1,804	6	71	2,002
given()	223	0	0	0	223
doThrow()	0	26	1	0	27
doAnswer()	0	5	0	0	5
doNothing()	0	0	0	0	0
doReturn()	0	10	0	1	11
verifyZeroInteractions()	0	83	1	5	89
verifyNoMoreInteractions()	4	61	2	0	67
thenReturn()	30	1,162	113	160	1,465
thenThrow()	1	10	7	0	18
Deleted calls to API					
verify()	646	2,141	110	250	3,147
when()	125	3,651	327	607	4,710
mock()	221	2,397	87	206	2,911
given()	136	0	0	1	137
doThrow()	9	31	4	0	44
doAnswer()	0	2	1	0	3
doNothing()	0	11	3	5	19
doReturn()	3	37	7	5	52
verifyZeroInteractions()	8	141	3	2	154
verifyNoMoreInteractions()	11	162	1	2	176
thenReturn()	113	3,269	309	557	4,248
thenThrow()	1	31	4	3	39

a) Mocks that changed due to changes in the production code. With our analysis, we observe two different types of changes that happen in production code that induce mocks to change: 'changes in the production class API' and 'changes in the internal implementation of the class.'

The former happens when the API of the production class being mocked changes in any way: the return type of the method is changed (13% of the cases), the number of parameters received by the method is changed (27%), the method is renamed (30%), or the entire class is either renamed or refactored (30%).

As discussed during the interviews with developers, a major challenge using mocks is correctly handling the coupling between production and test code. Our manual analysis corroborates the presence of this challenge: Indeed, we note how a change—even if minor, such as a renaming—in a production class can affect *all* its mocks in the test code. Interestingly, Sonarqube is the project in which this happens the most.

Concerning 'changes in the internal implementation of the class,' we found cases in which developers changed the internal encapsulated details of how a method works from the inside and this induced the mock to change accordingly. We observe this phenomenon in 73% of cases. Besides, we also observe classes moving away from a method and replacing it for another one (19%) and even production classes being completely replaced by others (8%), which then led to changes in the way the class/method is mocked in the test.

b) Mocks that changed during test evolution. Changes in the test code itself, not related to production code, are one of the main reason for mock objects to change. Among all the changes related to test code, we find that Mockito APIs change because of test refactoring (63%). More specifically, we observed mocks being changed due to the field or variable that hold their instances to be renamed, the *static import* of the Mockito's API so that the code becomes less noisy, and general refactoring on the test method. We also observed mocks being changed due to improvements that developers make in the test itself (32%), *e.g.*, testing different inputs and corner cases.

Interestingly, test refactorings that involve mocks happen more often in Alura, Spring, and VRaptor than in Sonarqube.

Finally, these results are in line with what we observed in RQ_3: Developers considered unstable dependencies, *i.e.*, maintaining the behavior of the mock compatible with the behavior of the original class, as a challenge.

> ***RQ_5.*** *Lines involving mocks change often. Test refactoring (not related to the mocks themselves), changes to the mocked production class, and changes to the internal implementation of the mocked class, are the most frequent reasons that induce a mock to change.*

Table 5.7: Classification of the changes to Mockito APIs.

Type of change	Spring framework	Sonarqube	VRaptor	Alura	Total
Test code related	57	20	42	45	164
Production class' API	19	33	36	25	113
Production class' internal implementation details	8	22	15	14	59

5.5 Discussion

In this section, we present the results of a debate about our findings with a core developer from Mockito. Next, we provide an initial quantitative evaluation of the mocking practices that emerged in our results and how much they apply to the systems under study.

5.5.1 Discussing with a developer from Mockito

To get an even deeper understanding of our results and challenge our conclusions, we interviewed a developer from Mockito, showing him the findings and discussing the challenges. We refer to him as D4.

D4 agreed on the findings regarding what developers should mock: According to him, databases and external dependencies should be mocked when developers do not test the integration itself, while Java libraries and data holders classes should never be mocked instead. Furthermore, D4 also approved what we discovered regarding mocking practices. He affirmed that a good practice is to mock interfaces instead of real classes and that developers should not mock the unit under test. When we argued whether Mockito could provide a feature to ease the mocking process of any of the analyzed categories (Figure 5.3), he stated: *"If someone tells us that s/he is spending 100 boiler-plate lines of code to mock a dependency, we can provide a better way to do it.* [...] *But for now, I can not see how to provide specific features for databases and web services, as Mockito only sees the interface of the class and not its internal behavior."*

After, we focused on the challenges, as we conjecture that it is the most important and useful part for practitioners and future research and that his experience can shed light on them. D4 agreed with all the challenges specified by our respondents. When discussing how Mockito could help developers with all the coupling challenges (unstable dependencies, highly coupled classes), he affirmed that the tool itself can not help and that the issue should be fixed in the production class: *"When a developer has to mock a lot of dependencies just to test a single unit, he can do it! However, it is a big red flag that the unit under test is not well designed."*. This reinforces the relationship between the excessive use of mocks and code quality.

When we discussed with him about a possible support for legacy systems in Mockito, D4 explained that Mockito developers have a philosophical debate internally: They want to keep a clear line of what this framework should and should not do. Not supported features such as the possibility of mocking a static method would enable developers to test their legacy code more efficiently. However, he stated: *"I think the problem is not adding this feature to Mockito, probably it will require just a week of work, the problem is: should*

we really do it? If we do it, we allow developers to write bad code." Indeed, mock proponents often believe that making use of static methods is a bad practice. Static methods cannot be easily mocked. In Java, mock frameworks dynamically create classes at runtime that either implement an interface or inherit from some base class, and implement/override its methods. Since it is impossible to override static methods in Java, a mock framework that wants to support such feature would have to either modify the bytecode of a class at runtime or replace the JVM's default classloader during the test execution.[17] As a consequence, static methods cannot be easily replaced by a mock implementation during a test; therefore, whenever a class invokes a static method developers have less control on their tests (regardless of whether this static method is part of the class under test or of an external class). Because of this limitation, mock proponents often suggest developers to write wrappers around static methods to facilitate testing (*e.g.*, a class `Clock` containing a `now()` instance method that wraps Java's `Calendar.getInstance()`).[18]

He also said that final classes can be mocked in Mockito 2.0; interestingly, the feature was not motivated by a willingness to ease the testing of legacy systems, but by developers using Kotlin language [19], in which every class is final by default.

To face the challenge of getting started with mocks, D4 mentioned that Mockito documentation is already extensive and provides several examples of how to better use the framework. However, according to him, knowing what should be mocked and what should not be mocked comes with experience.

5.5.2 Relationship between mocks and code quality

A recurrent topic throughout our interviews and surveys was about a possible relationship between the usage of mocks and the code quality of the mocked class. In other words, when classes are too coupled or complex, developers might prefer to mock their behavior instead of using their concrete implementation during the tests.

In this section, we take a first step towards the understanding of this relationship, by means of analyzing the code quality metrics of mocked/not mocked classes.

We take into account four metrics: CBO (Coupling between objects), McCabe's complexity [67], LOC (Lines of Code), NOM (Number of methods). We choose these metrics since they have been widely discussed during the interviews and, as pointed out during the surveys, developers mock when classes are very coupled or difficult to set up. In addition, CK metrics have proven to be useful in different predicting tasks, such as bug prediction [157] and class testability [158].

To obtain software code quality metrics, we used the tool CK [20]: we chose this tool because (i) it can calculate code metrics in Java projects by means of static analysis (i.e., no need for compiled code), and (ii) authors were already familiar with this tool [127].

We linked the output of our tool MockExtractor and CK: for each production class in the four systems, we obtained all the necessary code metrics and the number of times the class was/not mocked. With the metrics value for each production class, we compare

[17] As an example, Powermock (a Java framework that can mock static methods) makes use of both bytecode manipulation and a custom classloader. More information can be found at the project's official page: https://github.com/powermock/powermock. Last access in July, 2018.

[18] The Clock wrapper example is taken from the Alura project.

[19] https://kotlinlang.org

[20] https://github.com/mauricioaniche/ck

the values from classes that are mocked with the values from classes that are not mocked. In general, as a class can be mocked and not mocked multiple times, we apply a simple heuristic to decide in which category it should belong: If the class has been mocked more than 50% of the times, we put it in the 'mocked' category, and vice-versa (*e.g.*, if a class has been mocked 5 times and not mocked 3 times, it will be categorized as 'mocked').

Furthermore, to control for the fact that classes may be tested a different number of times (which could influence the developer's mocking strategy), we divided the classes into four categories, according to the number of times they were tested. To choose the categories, we used the 80th, 90th and 95th percentiles, as done by previous research [116, 127]. The used thresholds where: Low ($x \leq 4$), Medium ($4 < x \leq 7$), High ($7 < x \leq 12$), and Very High ($x \geq 12$).

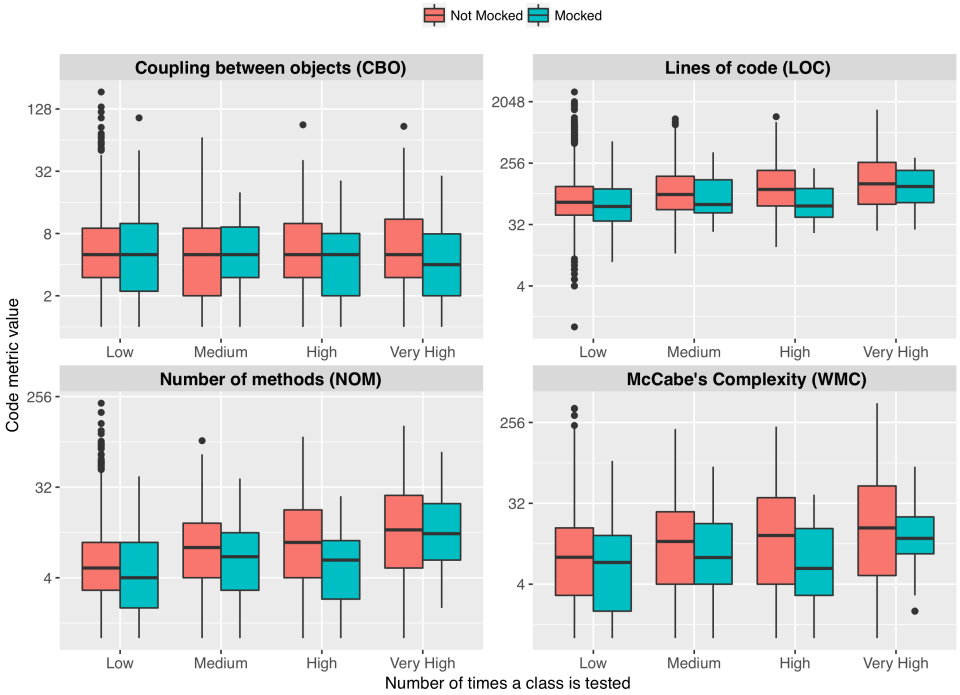


Figure 5.6: Code quality metrics comparison between mocked/not mocked classes (log scale).

To compare code metrics of mocked and not mocked production classes, we use the Wilcoxon rank sum test [107] (with confidence level of 95%) and Cliff's delta [109] to measure the effect size. We choose Wilcoxon because our distribution is not normal (we checked by both inspecting the histogram as well by running Shapiro-Wilk test [159]) and because it is a non-parametric test (does not have any assumption on the underlying data distribution).

In Figure 5.6, we show the results of the comparison between code metrics in mocked/not mocked classes. For every metric, we present the difference between mocked and not mocked classes in the 4 categories. To better present the results, we use a log scale on the

y-axis.

As a result, we see that both mocked and non mocked classes are similar in all the metrics. From Wilcoxon rank sum test and the effect size, we observed that almost in all the cases the overall difference is negligible (Wilcoxon p-value<0.001, Cliff's Delta<−0.12). There are however some exceptions: in terms of LOC, we can notice that in high and very high tested classes, the value for the mocked ones is slightly higher (Cliff's Delta=−0.37). Regarding complexity, again for high and very high tested classes, the value for the mocked category is higher than for the non mocked category (Cliff's Delta=−0.29).

Interestingly, these results *are not* in line with what we discovered during the interviews and surveys: indeed, even though developers said that they prefer to mock complex or highly coupled classes, it seems not to be the case if we look at code metrics. We conjecture that the chosen code metrics are not enough to explain when a class should or should not be mocked. Future work should better understand how code metrics are related to mocking decisions.

5.5.3 To Mock or Not to Mock: The Trade-offs

We started this chapter by stating the trade-off that developers have to make when deciding to use mocks in their tests: *by testing all dependencies together, developers gain realism. By simulating its dependencies, developers gain focus.* Moreover, our first paper was called "To Mock or Not to Mock?".

Indeed, our study has been around the decision process that developers go through when deciding whether to use mocks in their tests. As with most decisions in software engineering, deciding whether to use mocks comes with pros and cons. The trade-offs in these different decisions were an orthogonal topic throughout our research findings. In this section, we group and highlight them.

Firstly, our interviewees (technical leaders) were all aware that their tests become less realistic with mocks and that, as a consequence, (an important) part of their system was not being actually tested. To tackle this problem, all the interviewees have been combining different levels of testing in their test suites: A class might be mocked throughout the tests suites to enable other classes to be unit tested; however, that same class, when being the class under test, is tested via integration tests. As an example, the Alura technical leader explained that their Data Access Objects (DAOs) are often mocked throughout their test suite; however, all their DAOs have dedicated integration tests that aim at exercising the SQL query in a real external database. We made the same observation when exploring Sonarqube's test suite.[21]

This is a trade-off they have been making: they mock a certain dependency, so that tests are easier to be written. However, they pay the price of writing integration tests for these dependencies later on, to make sure these dependencies really work as expected.

Secondly, the interviewees made effort to have their systems "easy to be tested". This can mean two different things: designing classes in such a way that mocking is possible, and avoiding complex classes that would require too much mocking. For the former, we see how developers have been taking class design decisions "for the sake of testability". The Spring developer affirmed to create interfaces that represent side effects, such as HTTP

[21] The `org.sonar.db.user.UserDao` is an example of such class. The DAO is mocked throughout the test suite, and the DAO itself is tested by means of an integration test (see `UserDaoTest` class).

calls (which are naturally harder to be tested in isolation), just for the sake of simplifying the test. As exemplified before, Alura has a "Clock" abstraction to ease the simulation of different dates and times. The same pattern happens in Sonarqube[22]. This is another trade-off developers are making when it comes to testability and mocking: on one hand, the system gets more complex (in fact, new abstractions are introduced to the code base), on the other hand writing tests gets easier.

Concerning the latter, we also observed how simplicity plays a great role in how interviewees judge the difficulty of testing a class. For example, when our interview comes to the point where we discussed domain objects, we could see that all interviewees only thought mocking this type of classes when too complex. In addition, while we could not observe any relationship between CK metrics and the usage of metrics, an important perception of our participants was that complexity means difficulty in testing, which implies in the (possibly excessive) usage of mocks.

Finally, our interviewees are aware of the coupling they introduce when using mock objects and the fragility this brings to their test suites. In other words, they know that whenever they make use of a mock, that mock might be sensitive to changes in the original class. Such changes are naturally propagated to their test suites, which will then require developers to spend time in fixing them. Apparently, this is a trade-off they currently choose to pay: more testability comes with the price of a higher fragility in their test code.

Overall developers are aware of the positive and the negative aspects of using mock objects in their test suites. The question we raise for future researchers is: *what we can do to reduce the impact of the negative ones?*

5.5.4 Threats to Validity

In this section, we pose possible threats to the validity of our results as well as the actions we took to mitigate them.

Construct validity. Threats to *construct validity* concern our research instruments:

1. We develop and use MockExtractor to collect dependencies that are mocked in a test unit by means of static code analysis. As with any static code analysis tool, MockExtractor is not able to capture dynamic behavior (*e.g.*, mock instances that are generated in helper classes and passed to the test unit). In these cases, the dependency would have been considered "not mocked". We mitigate this issue by (1) making use of a large random samples in our manual analysis, and (2) manually inspecting the results of MockExtractor in 100 test units, in which we observed that such cases never occurred, thus giving us confidence regarding the reliability of our data set.

2. In the first part of the study, as only a single researcher manually analyzes each class and there could be divergent opinions despite the discussion mentioned above, we measured their agreement. Each researcher analyzed 25 instances that were made by the other researcher in both of his two projects, totaling 100 validated instances as seen in Figure 5.1, Point 2. The final agreement on the seven categories was 89%.

[22]See `PurgeProfiler` and its `Clock` internal class

3. In the second part of the study, namely the evolution of mocks, we devised a tool that extracts source code information about the changes that mock objects suffer throughout history. To keep track of the changes, we had to link source code lines in the old and new version of the *source code diff* that Git provides between two commits; such link is not readily available. To that aim, as we explain in Section 5.3, we apply cosine similarity to determine whether two lines are the same. Such heuristic may be prone to errors. In order to mitigate this thread, we determined the threshold for the cosine similarity after experimenting it in 75 randomly selected real changed lines from the four software systems. The chosen threshold achieves a precision of 73.2%. Although we consider the precision to be enough for this study, future research needs to be conducted to determine line changes in source code diffs.

4. Our tool detects changes of 13 Mockito APIs. These 13 APIs happen to be in Mockito since its very first version and past literature [142] shows that they are the most used APIs. Choosing these 13 APIs did not allow us to investigate "adoption patterns" (i.e., how developers take advantage of a newly introduced mocking API). Indeed, Mockito development history shows that new APIs are constantly being added, and thus, we leave as future work to explore their adoption.

5. As explained in Section 5.3, the first two authors manually classified the types of technical debt. Since this process was done simultaneously (the authors were seated in the same room next to each other) and they were discussing each technical debt, no validation of agreement was needed.

6. Our tool also is able to statically detect changes in lines that involve Mockito API, *e.g.*, `verify(mock).action()`. In practice, different implementation strategies may result in different results. As an example, if a test class A contains one `verify` line in each test (thus, several `verifys` in the source code), and another test class B encapsulates this call in a private method (thus, just a single `verify` call in the source code), the change analysis in both classes will yield different results. However, as our analysis is performed in scale (*i.e.*, we analyzed *all* the commits in the main branch of the four systems), we conjecture that such small differences do not have a large impact on the implications of our study.

7. In Section 5.5.2, we investigated the relation between mocks and code quality. In the comparison between mocked and not mocked classes, we controlled for the fact that classes may be tested a different number of times. To this aim, we divided the classes into four categories (Low, Medium, High, Very High), according to the number of times they were tested. To choose the threshold of the categories, we used the 80th, 90th and 95th percentiles. Even though these percentiles have been already used in previous research [116, 127], different thresholds may lead to different results. It is in our future agenda to better investigate the relation between mocks and code quality, with a deeper analysis on the characteristics of the most mocked classes.

Internal validity. Threats to *internal validity* concern factors we did not consider that could affect the variables and the relations being investigated:

1. We performed manual analysis and interviews to understand why certain dependencies are mocked and not mocked. A single developer does not know all the implementation decisions in a software system and may think and behave differently from the rest of the team. Hence, developers may wrongly choose to mock/not mock a class. We tried to mitigate this issue in several ways: (1) during interviews, by explicitly discussing both their point of view as well as the "rules" that are followed by the entire team, (2) and by presenting the results of RQ_1 and asking them to help us interpret it, with the hope that this would help them to see the big picture of their own project; (3) regarding the manual analysis, by analyzing large and important OSS projects with stringent policies on source code quality, and (4) by quantitatively analyzing a large set of production classes (a total of 38,313 classes) and their mocking decisions.

2. During the interview, their opinions may also be influenced by other factors, such as current literature on mocking (which could may have led them to social desirability bias [160]) or other projects that they participate in. To mitigate this issue, we constantly reminded interviewees that we were discussing the mocking practices specifically of their project. At the end of the interview, we asked them to freely talk about their ideas on mocking in general.

3. To perform the manual analysis in RQ_5, we randomly selected 100 changes of each system (which gives a CL=95%, CI=10). During the analysis, changes in larger commits may appear more often than changes in smaller commits, *e.g.*, several changes in mock objects may be related to the same large refactoring commit. We observed such effect particularly during the analysis of VRaptor. Due to the amount of changes we analyzed in the four systems, we do not expect significant variation in the results. Nevertheless, it is part of our future work to perform stratified sampling and compare the results.

External validity Threats to *external validity* concern the generalization of results:

1. Our sample contains four Java systems (one of them closed source), which is small compared to the overall population of software systems that make use of mocking. We reduce this issue by collecting the opinion of 105 developers from a variety of projects about our findings. Further research in different projects in different programming languages should be conducted.

2. Similarly, we are not able to generalize the results we found regarding the evolution of the mocks. As a way to reduce the threat, the four analyzed systems present different characteristics and focus on different domains. Future replication should be conducted to consolidate our results.

3. We do not know the nature of the population that responded to our survey, hence it might suffer from a self-selection bias. We cannot calculate the response rate of our survey; however, from the responses we see a general diversity in terms of software development experience that appears to match in our target population.

5.6 Related Work

In this section, we present related work on: (1) empirical studies on the usage of mock objects, (2) studies on test evolution and test code smells (and the lack of mocking in such studies), (3) how automated test generation tools are using mock objects to isolate external dependencies, and (4) the usage of pragmatic unit testing and mocks by developers and their experiences.

Empirical studies on the usage of mock objects. Despite the widespread usage of mocks, very few studies analyzed current mocking practices. Mostafa *et al.* [142] conducted an empirical study on more than 5,000 open source software projects from GitHub, analyzing how many projects are using a mocking framework and which Java APIs are the most mocked ones. The result of this study shows that 23% of the projects are using at least one mocking framework, Mockito being the most widely used (70%). In addition, software testers seem to mock only a small portion of all dependency classes of a test class. On average, about 17% of dependency classes are mocked by the software testers. This is also observed in the number of mock objects in test classes: 45% of test files contain just a single mock, and 21% contain just two mocks; only 14% contain five or more mock objects. Their results also show that about 39% of mocked classes are library classes. This implies that software testers tend to mock more classes that belong to their own source code, when compared with library classes. In terms of API usage, `Mockito.verify()` (used to perform assertions in the mock object), `Mockito.mock()` (used to instantiate a mock), and `Mockito.when()` (used to define the behavior of the mock) are by far the most used methods. This is in line with the results for our RQ_5. They also observed similar results for EasyMock (the second most popular mock framework in their study).

Marri *et al.* [143] investigated the benefits as well as challenges of using mock objects to test file-system-dependent software. Their study identifies the following two benefits: 1) mock objects enable unit testing of the code that interacts with external APIs related to the environment such as a file system, and 2) allow the generation of high-covering unit tests. However, according to the authors, mock objects can cause problems when the code under test involves interactions with multiple APIs that use the same data or interact with the same environment.

Test evolution and code smells (and the lack of mocking). Studies that focus on the evolution of test code, test code smells, and test code bugs have been conducted. However, they currently do not take mocks as a perspective, which should be seen as suggestions for future work. Vahabzadeh *et al.* [112] mined 5,556 test-related bug reports from 211 projects from the Apache Software Foundation to understand bugs in test code. Results show that false alarms are mostly caused by semantic bugs (25%), flaky tests (21%) environment (18%), and resource handling (14%). Among the environmental alarms, 61% are due to platform-specific failures, caused by operating system differences. Authors did not report any bugs related to mock objects, which leaves us to conclude that either these tests did not make use of mock objects, or mocks were not taken into account during their analysis.

Zaidman *et al.* [56] investigated how test and production code co-evolve in both open source and industrial projects. Authors found that production code and test code are usually modified together, that there is no clear evidence of a testing phase preceding a release, and that only one project in their studied sample used Test-Driven Development (they approximated it by looking to tests and production files committed together). In another

study, Zaidman *et al.* [161] performed a two-group controlled experiment involving 42 participants with the focus on investigating whether having unit tests available during refactoring leads to quicker refactorings and higher code quality. Results, however, indicate that having unit tests available during refactoring does not lead to quicker refactoring or higher-quality code after refactoring. Although the system used in the experiment made use of mocks, authors did not use mocks as control, and thus, the paper does not shed light on the relationship between mock objects and test refactoring.

Van Deursen *et al.* [2] coined the term *test smells* and defined the first catalog of eleven poor design solutions to write tests, together with refactoring operations aimed at removing them. Such a catalog has been then extended more recently by practitioners, such as Meszaros who defined 18 new test smells [7]. Although the catalog contains the `Slow tests` smell, which a solution could be the use of mock objects, there are no smells specific to the usage of mocks.

Such investigation can be important to the community, as it is known that test smells happen in real systems and have a negative impact on their maintenance. Greiler *et al.* [8, 9] showed that test smells affecting test fixtures frequently occur in industry. Motivated by this prominence, Greiler *et al.* presented TestHound, a tool able to identify fixture-related test smells such as *General Fixture* or *Vague Header Setup* [8]. Van Rompaey *et al.* [10] also proposes detection strategies for *General Fixture* and *Eager Test*, although their empirical study shows that the common often misses smelly instances.

Bavota *et al.* [11] studied the diffusion of test smells in 18 software projects and their effects on software maintenance. As a result, authors found that 82% of test classes are affected by at least one test smell. Interestingly, the same problem happens in test cases that are automatically generated by testing tools [86]. In addition, Bavota *et al.* [11] show that the presence of test smells has a strong negative impact on the comprehensibility of the affected classes. Tufano *et al.* [12] also showed that test smells are usually introduced during the first commit involving the affected test classes, and in almost 80% of the cases they are never removed, essentially because of poor awareness of developers.

Automatic Test Generation and Mocks. Taneja *et al.* [162] stated that automatic techniques to generate tests face two significant challenges when applied to database applications: 1) they assume that the database that the application under test interacts with is accessible, and 2) they usually cannot create necessary database states as a part of the generated tests. For these reasons, authors proposed an "Automated Test Generation" for Database Applications using mock objects, demonstrating that with this technique they could achieve better test coverage.

Arcuri *et al.* [147] applied bytecode instrumentation to automatically separate code from any external dependency. After implementing a prototype in EvoSuite [163] that was able to handle environmental interactions such as keyboard inputs, file system, and several non-deterministic functions of Java, authors show that EvoSuite was able to significantly improve the code coverage of 100 Java projects; in some cases, the improvement was in the order of 80% to 90%.

Another study, also by Arcuri *et al.* [146], focused on extending the EvoSuite unit test generation tool with the ability to directly access private APIs (via reflection) and to create mock objects using Mockito. Their experiments on the SF110 and Defects4J benchmarks confirm the anticipated improvements in terms of code coverage and bug finding, but also

confirm the existence of false positives (due to the tests that make use of reflection, and thus, depend on specificities of the production class, *e.g.*, a test accessing a private field will fail if that field is later renamed).

Finally, Li *et al.* [164] proposed a technique that combines static analysis, natural language processing, backward slicing, and code summarization techniques to automatically generate natural language documentation of unit test cases. After evaluating the tool with a set of developers, authors found out that the descriptions generated by their tool are easy to read and understand. Interestingly, a developer said: *"mock-style tests are not well described."*, suggesting that the tool may need improvement in tests that make use of mock objects.

Pragmatic unit testing and mocks. Several industry key leaders and developers affirm that the usage of mock objects can bring benefits to testing. Such experience reports call for in-depth, scientific studies on the effects of mocking.

Tim *et al.* [4], for example, in their chapter on a book about Extreme Programming, stated that using Mock Objects is the only way to unit test domain code that depends on state that is difficult or impossible to reproduce. They show that the usage of mocks encourages better-structured tests and reduces the cost of writing stub code, with a common format for unit tests that is easy to learn and understand.

Karlesky *et al.* [165], after their real-world experience in testing embedded systems, present a holistic set of practices, platform independent tools, and a new design pattern (Model Conductor Hardware - MCH) that together produce: good design from tests programmed first, logic decoupled from hardware, and systems testable under automation. Interestingly, the authors show how to mock hardware behavior to write unit tests for embedded systems.

Similarly, Kim *et al.* [166] stated that unit testing within the embedded systems industry poses several unique challenges: software is often developed on a different machine than it will run on and it is tightly coupled with the target hardware. This study shows how unit testing techniques and mocking frameworks can facilitate the design process, increase code coverage and the protection against regression defects.

5.7 Conclusion

Mocking is a common testing practice among software developers. However, there is little empirical evidence on how developers actually apply the technique in their software systems. We investigated *how* and *why* developers currently use mock objects. To that end, we studied three OSS projects and one industrial system, interviewed three of their developers, surveyed 105 professionals, and discussed the findings with a main developer from the leading Java mocking framework.

Our results show that developers tend to mock dependencies that make testing difficult, *i.e.*, classes that are hard to set up or that depend on external resources. In contrast, developers do not often mock classes that they can fully control. Interestingly, a class being slow is not an important factor for developers when mocking. As for challenges, developers affirm that challenges when mocking are mostly technical, such as dealing with unstable dependencies, the coupling between the mock and the production code, legacy systems, and hard-to-test classes are the most important ones. Studying the evolution of

mocks, we found that they are generally introduced at the inception of test classes and tend to stay within these classes for the entire lifetime of the classes. Mocks changes in an equally frequent way for changes to the production code that they simulate and for changes to the test code (*e.g.*, refactoring) that use them.

Our future agenda includes understanding the relationship between code quality metrics and the use of mocking practices, investigating the reasons behind mock deletions, and analyzing adoption patterns as well as the differences of mock adoptions in dynamic languages.

6

Information Needs in Contemporary Code Review

Contemporary code review is a widespread practice used by software engineers to maintain high software quality and share project knowledge. However, conducting proper code review takes time and developers often have limited time for review. In this chapter, we aim at investigating the information that reviewers need to conduct a proper code review, to better understand this process and how research and tool support can make developers become more effective and efficient reviewers.

Previous work has provided evidence that a successful code review process is one in which reviewers and authors actively participate and collaborate. In these cases, the threads of discussions that are saved by code review tools are a precious source of information that can be later exploited for research and practice. In this chapter, we focus on this source of information as a way to gather reliable data on the aforementioned reviewers' needs. We manually analyze 900 code review comments from three large open-source projects and organize them in categories by means of a card sort. Our results highlight the presence of seven high-level information needs, such as knowing the uses of methods and variables declared/modified in the code under review. Based on these results we suggest ways in which future code review tools can better support collaboration and the reviewing task.

Preprint: https://doi.org/10.5281/zenodo.1405894,

Data and Materials: https://doi.org/10.5281/zenodo.3698929.

This chapter has been published as *Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information Needs in Contemporary Code Review. Proceedings of the ACM Human-Computer Interaction, (CSCW), nov 2018.*[167]

6.1 Introduction

Peer code review is a well-established software engineering practice aimed at maintaining and promoting source code quality, as well as sustaining development community by means of knowledge transfer of design and implementation solutions applied by others [21]. Contemporary code review, also known as *Modern Code Review* (MCR) [16, 21], represents a lightweight process that is (1) informal, (2) tool-based, (3) asynchronous, and (4) focused on inspecting new proposed code changes rather than the whole codebase [27]. In a typical code review process, developers (the *reviewers*) other than the code change author manually inspect new committed changes to find as many issues as possible and provide feedback that needs to be addressed by the author of the change before the code is accepted and put into production [168].

Modern code review is a collaborative process in which reviewers and authors conduct an asynchronous online discussion to ensure that the proposed code changes are of sufficiently high quality [21] and fit the project's direction [169] before they are accepted. In code reviews, discussions range from low-level concerns (*e.g.*, variable naming and code style) up to high-level considerations (*e.g.*, fit within the scope of the project and future planning) and encompass both functional defects and evolutionary aspects [19]. For example a reviewer may ask questions regarding the structure of the changed code [170] or clarifications about the rationale behind some design decisions [171], another reviewer may respond or continue the thread of questions, and the author can answer the questions (*e.g.*, explaining the motivation that led to a change) and implement changes to the code to address the reviewers' remark.

Even though studies have shown that modern code review has the *potential* to support software quality and dependability [16, 32, 172], researchers have also provided strong empirical evidence that the outcome of this process is rather erratic and often unsatisfying or misaligned with the expectations of participants [19, 21, 173]. This erratic outcome is caused by the cognitive-demanding nature of reviewing [35], whose outcome mostly depends on the time and zeal of the involved reviewers [16].

Based on this, a large portion of the research efforts on tools and processes to help code reviewing is explicitly or implicitly based on the assumption that reducing the *cognitive load* of reviewers improves their code review performance [35]. In the current study, we continue on this line of better supporting the code review process through the reduction of reviewers' cognitive load. Specifically, *our goal is to investigate the information that reviewers need to conduct a proper code review.* We argue that—if this information would be available at hand—reviewers could focus their efforts and time on correctly evaluating and improving the code under review, rather than spending cognitive effort and time on collecting the missing information. By investigating reviewers' information needs, we can better understand the code review process, guide future research efforts, and envision how tool support can make developers become more effective and efficient reviewers.

To gather data about reviewers' information needs we turn to one of the collaborative aspects of code review, namely the discussions among participants that happen during this process. In fact, past research has shown that code review is more successful when there is a functioning collaboration among all the participants. For example, Rigby *et al.* reported that the efficiency and effectiveness of code reviews are most affected by the amount of review participation [24]; Kononenko *et al.* [174] showed that review participation metrics

are associated with the quality of the code review process; McIntosh *et al.* found that a lack of review participation can have a negative impact on long-term software quality [32, 33]; and Spadini *et al.* studied review participation in production and test files, presenting a set of identified obstacles limiting the review of code [20]. For this reason, from code review communication, we expect to gather evidence of reviewers' information needs that are solved through the collaborative discussion among the participants.

To that end, we consider three large open-source software projects and manually analyze 900 code review discussion threads that started from a reviewer's question. We focus on what kind of questions are asked in these comments and their answers. As shown in previous research [171, 175–177], such questions can implicitly represent the information needs of code reviewers. In addition, we conduct four semi-structured interviews with developers from the considered systems and one focus group with developers from a software quality consultancy firm, both to challenge our outcome and to discuss developers' perceptions. Better understanding what reviewers' information needs are can lead to reduced cognitive load for the reviewers, thus leading, in turn, to better and shorter reviews. Furthermore, knowing these needs helps driving the research community toward the definition of methodologies and tools able to properly support code reviewers when verifying newly submitted code changes.

Our analysis led to seven high-level information needs, such as knowing the uses of methods and variables declared/modified in the code under review, and their analysis in the code review lifecycle. Among our results, we found that the needs to know (1) whether a proposed alternative solution is valid and (2) whether the understanding of the reviewer about the code under review is correct are the most prominent ones. Moreover, all the reviewers' information needs are replied to within a median time of seven hours, thus pointing to the large time savings that can be achieved by addressing these needs through automated tools. Based on these results, we discuss how future code review tools can better support collaboration and the reviewing task.

6.2 Related Work

Breu *et al.* [176] conducted a study—which has been a great inspiration to the current study we present here—on developers' information needs based on the analysis of collaboration among users of a software engineering tool (*i.e.*, issue tracking system). In their study, the authors have quantitatively and qualitatively analyzed the questions asked in a sample of 600 bug reports from two open-source projects, deriving a set of information needs in bug reports. The authors showed that active and ongoing participation were important factors needed for making progress on the bugs reported by users and they suggested a number of actions to be performed by the researchers and tool vendors in order to improve bug tracking systems.

Ko *et al.* [175] studied information needs of developers in collocated development teams. The authors observed the daily work of developers and noted the types of information desired. They identified 21 different information types in the collected data and discussed the implications of their findings for software designers and engineers. Buse and Zimmermann [177] analyzed developers' needs for software development analytics: to that end, they surveyed 110 developers and project managers. With the collected responses, the authors proposed several guidelines for analytics tools in software develop-

ment.

Sillito *et al.* [178] conducted a qualitative study on the questions that programmers ask when performing change tasks. Their aim was to understand what information a programmer needs to know about a code base while performing a change task and also how they go about discovering that information. The authors categorized and described 44 different kinds of questions asked by the participants. Finally, Herbsleb *et al.* [179] analyzed the types of questions that get asked during design meetings in three organizations. They found that most questions concerned the project requirements, particularly what the software was supposed to do and, somewhat less frequently, scenarios of use. Moreover, they also discussed the implications of the study for design tools and methods.

The work we present in this chapter is complementary with respect to the ones discussed so far: indeed, we aim at making a further step ahead investigating the information needs of developers that review code changes with the aim of deepening our understanding of the code review process and of leading to future research and tools to better support reviewers in conducting their tasks.

6.3 Methodology

The *goal* of our study is to increase our empirical knowledge on the reviewers' needs when performing code review tasks, with the *purpose* of identifying promising paths for future research on code review and the next generation of software engineering tools required to improve collaboration and coordination between source code authors and reviewers. The perspective is of researchers, who are interested in understanding what are the developers' needs in code review, therefore, they can more effectively devise new methodologies and techniques helping practitioners in promoting a collaborative environment in code review and reduce discussion overheads, thus improving the overall code review process.

Starting from a set of discussion threads between authors and reviewers, we start our investigation by eliciting the actual needs that reviewers have when performing code review:

- **RQ$_1$:** *What reviewers' needs can be captured from code review discussions?*

Specifically, we analyze the types of information that reviewers may need when reviewing, we compute the frequency of each need, and we challenge our outcome with developers from the analyzed systems and from an external company. Thus, we have three sub-questions:

- **RQ$_{1.1}$:** *What are the kinds of information code reviewers require?*
- **RQ$_{1.2}$:** *How often does each category of reviewers' needs occur?*
- **RQ$_{1.3}$:** *How do developers' perceive the identified needs?*

Once investigated reviewers' needs from the reviewer perspective, we further explore the collaborative aspects of code review by asking:

- **RQ$_2$:** *What is the role of reviewers' needs in the lifecycle of a code review?*

Specifically, we first analyze how much each reviewers' need is accompanied by a reply from the author of the code change: in other words, we aim at measuring how much authors of the code under review interact with reviewers to make the applied code change more comprehensible and ease the reviewing process. To complement this analysis, we evaluate the time required by authors to address a reviewer's need; also in this case, the goal is to measure the degree of collaboration between authors and reviewers. Finally, we aim at understanding whether and how the reviewers' information needs vary at different iterations of the code review process. For instance, we want to assess whether some specific needs arise at the beginning of the process (*e.g.*, because the reviewer does not have enough initial context to understand the code change) or, similarly, if clarification questions only appear at a later stage (*e.g.*, when only the last details are missing and the context is clear). Accordingly, we structure our second research question into three sub-questions:

- **RQ$_{2.1}$:** *What are the reviewers' information needs that attract more discussion?*
- **RQ$_{2.2}$:** *How long does it take to get a response to each reviewers' information need?*
- **RQ$_{2.3}$:** *How do the reviewers' information needs change over the code review process?*

The following subsections describe the method we use to answer our research questions.

6.3.1 Subject Systems

The first step leading to address our research goals is the selection of a set of code reviews that might be representative for understanding the reviewers' needs when reviewing source code changes. We rely on the well-known Gerrit platform,[1] which is a code review tool used by several major software projects. Specifically, Gerrit provides a simplified web based code review interface and a repository manager for Git.[2] From the open-source software systems using Gerrit, we select three: OpenStack,[3] Android,[4] and QT.[5] The selection was driven by two criteria: (i) These systems have been extensively studied in the context of code review research and have been shown to be highly representative of the types of code review done over open-source projects *et al.* [11, 32, 180]; (ii) these systems have a large number of active authors and reviewers over a long development history.

6.3.2 Gathering Code Review Threads

We automatically mine Gerrit data by relying on the publicly available APIs it provides. For the considered projects, the number of code reviews is over one million: this makes the

[1] https://www.gerritcodereview.com/
[2] https://git-scm.com/
[3] https://review.openstack.org/
[4] https://android-review.googlesource.com/
[5] https://codereview.qt-project.org

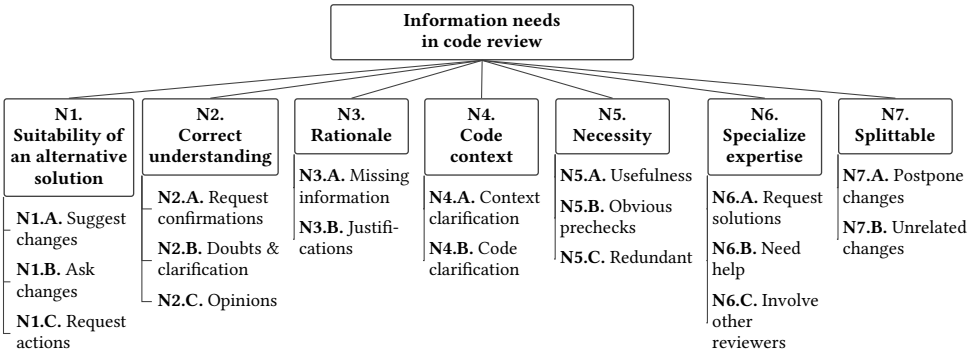


Figure 6.1: The taxonomy of reviewers' information needs that emerged from our analysis

manual analysis of all of them practically impossible. Thus, as done by Breu *et al.* [176], we select a random subset composed of 300 code reviews per project, for which we identify up to 1,800 messages (*i.e.*, we extract a total of 900 code review threads). Since we are interested in discussions, we take into account only closed code reviews by considering both merged and abandoned patches, while we do not consider recently opened or pending requests.

6

We detect reviewers' questions (considering the presence of a '?' sign) that start a discussion thread and we extract all the subsequent comments (made by the author, the reviewer, or other developers) in the whole thread. The considered threads refer to both patch sets and inline discussions.

For each identified *thread*, we store the following information:

- the *Gerrit id* of the code review;
- the *revision id* that identifies the patch set of a code review;
- the opening *question*, the *answers*, and the *source code* URL identifier of the change;
- the practitioner *role e.g.*, author or reviewer;
- the code review *status*, *i.e.*, whether it is merged or abandoned;
- the *size* of the thread counting the number of comments present into discussion;
- the creation and the update *time*.

We use the aforementioned pieces of information to answer our research questions as detailed in the following.

6.3.3 RQ_1 - Identifying the Reviewers' Needs from Code Review Discussions

To answer $\mathbf{RQ}_{1.1}$, we manually identify the reviewers' needs in code review by following a similar strategy as done in previous work on information needs [175–179]. Specifically,

we perform a card sorting method [133] that involves all the authors of this chapter (2 graduate students, 1 research associate, and 1 faculty member - who have at least seven years of programming experience). From now on, we refer to them as the *inspectors*. This method represents a well-established sorting technique that is used in information architecture with the aim of creating mental models and allowing the definition of taxonomies from input data [133]. In our case, it is used to organize code review threads into hierarchies and identify common themes. We rely on code review threads (*i.e.*, questions and answers) to better understand the meaning behind reviewers' questions that may implicitly define the reviewers' need. Finally, we apply an *open card sorting*: We have no predefined groups of reviewers' information needs, rather the needs emerge and evolve during the procedure. In our case, the process consists of the three iterative sessions described as follow.

Iteration 1: Initially, two inspectors (the first two authors of this chapter) independently analyze an initial set of 100 OpenStack code review threads each. Then, they open a discussion on the reviewers' needs identified so far and try to reach a consensus on the names and types of the assigned categories. During the discussion, also the other two inspectors participate with the aim of validating the operations done in this iteration and suggesting possible improvements. As an output, this step provides a draft categorization of reviewers' needs.

Iteration 2: The first two inspectors re-categorize the 100 initial reviewers' needs according to the decisions taken during the discussion; then, they use the draft categorization as a basis for categorizing the remaining set of 200 code review threads belonging to OpenStack. This phase is used for both assessing the validity of the categories emerging from the first iteration (by confirming some of them and redefining others) and for discovering new categories. Once this iteration is completed, all the four inspectors open a new discussion aimed at refining the draft taxonomy, merging overlapping categories or better characterizing the existing ones. A second version of the taxonomy is produced.

Iteration 3: The first two inspectors re-categorize the 300 code review threads previously analyzed. Afterwards, the first inspector classifies the reviewers' needs concerning the two remaining considered systems. In doing so, the inspector tries to apply the defined categories on the set of code review threads of Android and QT. However, in cases where the inspector cannot directly apply the categories defined so far, the inspector reports such cases to the other inspectors so that a new discussion is opened. Unexpectedly this event did not eventually happen in practice; in fact, the inspector could fit all the needs in the previously defined taxonomy, even when considering new systems. This result suggests that the categorization emerging from the first iterations reached a saturation [181], valid at least within the considered sample of threads.

Additional validation. To further check and confirm the operations performed by the first inspector, the third author of this chapter—who was only involved in the discussion of the categories, but not in the assignment of the threads into categories—independently analyzed all the code review threads belonging to the three considered projects. The inspector classified all the 900 threads according to the second

version of the taxonomy, as defined through iteration 2. The inspector did not need to define any further categories (thus suggesting that the taxonomy was exhaustive for the considered sample), however in six cases there were a disagreement between the category he assigned and the one assigned by the first author: as a consequence, the two authors opened a discussion in order to reach an agreement on the actual category to assign to those code review threads. Overall, the inter-rater agreement between this inspector and the first one, computed using the Krippendorff's k [182], was 98%.

Following this iterative process, we defined a hierarchical categorization composed of two layers. The top layer consists of *seven* categories, while the inner layer consists of 18 subcategories. Figure 6.1 depicts the identified top- and sub-categories. During the iterative sessions, ≈4% of the analyzed code review threads are discarded from our analysis since they do not contain useful information to understand the reviewers' needs. We assign these comments to four temporary sub-categories that indicate the reasons why they are discarded (e.g., they are noise or sarcastic comments), successively, we gathered these comments, in an additional top-category *Discarded*.

To answer $\mathbf{RQ}_{1.1}$, we report the reviewers' needs belonging to the categories identified in the top layer.

Subsequently, to answer $\mathbf{RQ}_{1.2}$ and understand how frequently each category of our needs appears, we verify how many information needs are assigned to each category. In this way, we can overview the most popular reviewers' needs when performing code review tasks. We answer this research question by presenting and discussing bar plots showing the frequency of each identified category.

To answer $\mathbf{RQ}_{1.3}$, we discuss the outcome of the previous sub-RQs with developers of the three considered systems and an external company. This gives us the opportunity to challenge our findings, triangulate our results, and complement our vision on the problem.

Table 6.1: Interviewees' experience (in years) and their working context.

ID	Years as developer	Years as reviewer	Working context
P_1	15	10	OpenStack
P_2	20	10	OpenStack
P_3	25	20	Qt
P_4	10	10	Android
FG_1	8	7	Company A
FG_2	10	10	Company A
FG_3	7	5	Company A

Interviews with reviewers from the subject systems. To organize the discussion with the developers of Android, OpenStack, and Qt, we use semi-structured interviews– a format that is often used in exploratory investigations to understand phenomena and seek new insights [183]. A crucial step in this analysis is represented by the recruitment strategy, *i.e.*, the way we select and recruit participants for the semi-structured interviews. With the aim of gathering feedback and opinions from developers having a solid experience with the code review practices of the considered projects, we select only developers

who had conducted at least 100 reviews[6] in their respective systems. Then, we randomly select 10 per system and invite them via email to participate in an online, video interview. Four experienced code reviewers accepted to be interviewed: two from OpenStack, one from Qt, and one from Android. The response rate achieved (17%) is in line with the one achieved by many previous works involving developers [57, 184, 185]. Table 6.1 summarizes the interviewees' demographic.

The interviews are conducted by the first two authors of this work via Skype. With the participants' consent all the interviews are recorded and transcribed for analysis. Each interview starts with general questions about programming and code reviews experience. In addition, we discuss whether the interviewees consider code reviews important, which tool they prefer, and generally how they conduct reviews. Overall, we organize the interview structure around five sections:

1. General information regarding the developer;
2. General perceptions on and experience with code review;
3. Specific information needs during code review;
4. Ranking of information needs during code review;
5. Summary.

The main focus regarding the information needs is centered around points 3 and 4: We iteratively discuss each of the categories emerged from our analysis (also showing small examples where needed). Afterwards, we discuss the following main questions with each interviewee:

1. What is your experience with <category>?
2. Do you think <category> is important to successfully perform a code review? Why?
3. Do you think current code review tools support this need?
4. How would you improve current tools?

Our goal with these questions is to allow us to better understand the relevance of each developer's need and whether developers feel it is somehow incorporated in current code review tools or, if not, how they would envision this need incorporated. Successively, we ask developers to rank the categories according to their perceived importance. Our goal is to understand what the interviewees perceive as the most important needs and why. To conclude the interview, the first two authors of this chapter summarize the interview, and before finalizing the meeting, these summaries are presented to the interviewee to validate our interpretation of their opinions.

Focus group with an external commercial company. While the original developers provide an overview of the information needs identified in the context of the systems analyzed in this study, our findings may not provide enough diversity. To improve this

[6]This minimum number of reviews to ensure an appropriate experience of the interviewees is aligned with the numbers used in previous studies on code review (*e.g.*, [21]).

aspect, we complement the aforementioned semi-structured interviews with an additional analysis targeting experts in assessing the source code quality of systems. In particular, we recruited three employees from a firm in Europe specialized in software quality assessments for their customers. The mission of the firm is the definition of techniques and tools able to diagnose design problems in the clients' source code, with the purpose of providing consultancy on how to improve the productivity of their clients' industrial developers. Our decision to involve these quality experts is driven by the willingness to receive authoritative opinions from professionals who are used to perform code reviews for their customers. The three participants have more than 15 years in assess code quality and more than 10 years of experience in code review.

In this case we proceed with a *focus group* [186, 187] because it better fits our methodology. Indeed, this technique is particularly useful when a small number of people is available for discussing about a certain problem [186, 187] and consists of the organization of a meeting that involves the participants and a moderator. The moderator starts the discussion by asking general questions on the topic of interest and then leaves the participants to openly discuss about it with the aim of gathering additional qualitative data useful for the analysis of the results. In the context of this chapter, the first two authors are the moderators in a meeting directly organized in the consultancy firm. The focus group is one hour long and the participants reflected on and discuss the information needs we identified and what are the factors influencing their importance. From this analysis, our aim is also to better understand the external validity of our taxonomy.

6.3.4 RQ_2 - On the role of reviewers' needs in the lifecycle of a code review

In the context of the second research question we perform a fine-grained investigation of the role of reviewers' needs in code review. We analyze which of them capture more replies, what is the time required for getting an answer, and whether reviewers' needs change throughout the iterations.

Specifically, we consider code review threads related to the same reviewer's need independently. Then, to answer $\mathbf{RQ}_{2.1}$ we computed the number of replies that each group received: this is a metric that represents how much in deep reviewers and authors should interact to be able to exchange the information necessary to address the code review. We do not assess the quality of the responses, since we aim at reporting quantitative observations on the number of answers provided by authors to a reviewer's need.

As for $\mathbf{RQ}_{2.2}$, this represents a follow-up of the previously considered aspect. Indeed, besides assessing the number of replies for each reviewers' need, we also measure the time (in terms of minutes) needed to get a response. This complementary analysis can possibly provide insights on whether certain needs require authors to spend more time to make their change understandable, thus providing information on the relative importance of each need which might be further exploited to prioritize software engineering research effort when devising and developing new techniques to assist code reviewers.

Finally, to answer $\mathbf{RQ}_{2.3}$ and understand how the reviewers' needs change over the code review iterations, we measure the number of times a certain need appears in each iteration of a code review. This analysis may possibly lead to observations needed by the research community to promptly provide developers with appropriate feedback during

the different phases of the code review process.

As a final step of our methodology, we compute pairwise statistical tests aimed at verifying whether the observations of each sub-research question are statistically significant. We apply the Mann-Whitney test [188]. This is a non-parametric test used to evaluate the null hypothesis stating that it is equally likely that a randomly selected value from one sample will be less than or greater than a randomly selected value from a second sample. The results are intended as statistically significant at α=0.05. We also estimate the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure for ordinal data [189]. We follow well-established guidelines to interpret the effect size values: negligible for $|d|$<0.10, small for 0.10≤ $|d|$ < 0.33, medium for 0.33 ≤ $|d|$ < 0.474, and large for $|d| \geq 0.474$ [189].

6.4 Results

In this section, we present and analyze the results of our study by research question.

6.4.1 RQ_1 - A Catalog of Reviewers' Information Needs

We report the results of our first research questions, which aimed at cataloging reviewers' information needs in code review and assessing their diffusion. For the sake of comprehensibility, we answer each sub-research question independently.

$RQ_{1.1}$: *What are the kinds of information code reviewers require?*

Following the methodology previously described (Section 6.3.3), we obtained 22 groups of reviewers' information needs. They were then clustered according to their intention into *seven* high-level categories that represent the classes of information needs associated with the discussion threads considered in our study. We describe each high-level category also including representative examples.

N1. Suitability of An Alternative Solution
This category emerged by grouping threads in which the reviewer poses a question to discuss options and alternative solutions to the implementation proposed by the author in the first place. The purpose is not only to evaluate alternatives but also to trigger a discussion on potential improvements. The example reported in the following reports a case where the reviewer starts reasoning on how much an alternative solution is suitable for the proposed code change.

> R: "... Since the change owner is always admin, this code might be able to move out of the loop? The following should be enough for this? [lines of code]"

N2. Correct understanding
In this category, we group questions in which the reviewers try to ensure to have captured the real meaning of the changes under review; in other words, this category refers to questions asked to get a consensus of reviewers' interpretation and to clarify doubts. This is more frequent when code comments or related documentation is missing. as reported in the example shown in the following.

> R: "This is now an empty heading ...Or do you feel it is important to point out that these are C++ classes?"
> A: "The entire page is split up into [more artifacts]. The following sections only refer to [one artifact]. I added a sentence introducing the section."

N3. Rationale
This category refers to questions asked to get missing information that may be relevant to justify why the project needs the submitted change set or why a specific change part was implemented/designed in a certain way. For example, a reviewer may request more details about the issue that the patch is trying to address. These details help the reviewer in better understanding whether the change fits with the project scope and style. For instance, in the example reported below the reviewer (R) asks why the author replaced a piece of code.

> R: "Can you explain why you replaced [that] with [this] and where exactly was failing?"

N4. Code Context
In this category, we grouped questions asked to retrieve information aimed at clarifying the context of a given implementation. During a code review, a reviewer has access to the entire codebase and, in this way, may reconstruct the invocation path of a given function to understand the impact of the proposed change. However, we observed that the reviewer needs contextual information to clarify a particular choice made by authors. These questions range from very specific (*i.e.*, aimed at understanding the code behavior) to more generic (*i.e.*, aimed at clarifying the context in which such code is executed). The author replies to such questions by providing additional explanations on the code change or contextual project details. For instance, let consider the thread reported below, where the Author (A) replies to the Reviewer (R) by pointing R to the file (and the line) containing the asked clarification.

> R: "In what situations would [this condition] be false, but not undefined?"
> A: "See [file], exactly in line [number], in this case the evaluation of the expression returns false."
> R: "It may be helpful to add a comment documenting these situations to avoid future regressions."

N5. Necessity
In this category, the reviewer needs to know whether a (part of) the change is really necessary or can be simplified/removed. For example, a reviewer may spot something that seems like a duplicated code, yet is unsure if whether the existing version is a viable solution or it should be implemented as proposed by the author. In the example below, the reviewer asks whether a certain piece of code could be removed.

> R: "Is this needed?"
> A: "I believe its only required if you have methods after the last enum value, but I generally add it regardless. We have a pretty arbitrary mix."

N6. Specialized Expertise
Threads belonging to this category regard situations in which a reviewer finds or feels there is a code issue, however, the reviewer's knowledge is not appropriate to propose a solution. In these cases, typically a reviewer asks other reviewers to step in and contribute with their specialized expertise. Sometimes, reviewers may ask the author to propose informal alternatives that may better address the found issue. The examples reported below show two cases where the reviewer encourages other developers to reason on how to fix an issue.

> R: "... Lars, Simon, any ideas? We really need to fix this for [the next release] and the time draws nigh"

> R: "I need a better way to handle this ...not a good idea to hard code digits in there. example also needs to be removed, its there just to make the tests pass."

N7. Splittable
For several reasons (including reducing the cognitive load of reviewers [190]), authors want to propose changes that are *atomic* and *self-contained* (*e.g.*, address a single issue or add a single feature). However, sometimes, what authors propose may be perceived by reviewers as something that can be addressed by different code changes, thus reviewed separately. For this reason, a reviewer needs to understand whether the split she has in mind can be done; based on this the reviewer asks questions aimed at finding practical evidence behind this idea. In other words, this category gathers questions proposed by reviewers who need to understand whether the proposed changes can be split into multiple, separated patches. For example, the thread below reports a question where the reviewer (R) asks the author about the possibility of splitting unrelated changes, but the feasibility of this split is not confirmed.

> R: "This looks like an unrelated change. Should it be in a separate commit?"
> A: "Actually its related. The input object is needed to log the delete options."
> R: "OK, I wasn't sure because in the previous version we don't pass 'force' into the method, but now we do pass it in via the 'input'."

In addition to the aforementioned categories, we found several cases in which the presence of a question or question mark did not correspond to a real information need, similarly to the aforementioned categorized information needs, we provide an example in the following.

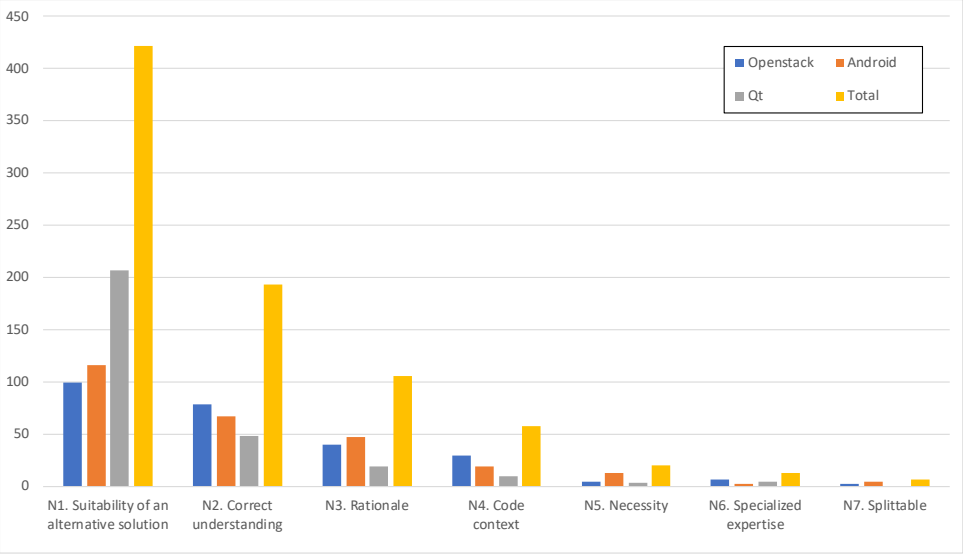


Figure 6.2: Distribution of reviewers' information needs across the considered systems.

> R: "I hate `name` as a name. What kind of name is this?"

> R: 'If you thought it was necessary to check `exe()` for errors, then why'd you leave out [another part] here? :)"

RQ$_{1.2}$: *How often does each category of reviewers' needs occur?*

Figure 6.2 depicts bar plots that show the distribution of each reviewers' need over the considered set of code review threads. The results clearly reveal that not all the information needs are equally distributed and highlight the presence of a particular type (*i.e.*, N1. 'Suitability of an alternative solution'), which has a way larger number of occurrences with respect to all the others (the result is consistent over the three considered systems). Thus, we can argue that one of the most useful tools for reviewers would definitively be one that allows them to have just-in-time feedback on the actual practicability of possible alternative solutions with respect to the one originally implemented by the authors of the committed code change.

The second most popular category is represented by 'Correct understanding' (N2), *i.e.*, questions aimed at assessing the reviewers' interpretation of the code change and to clarify doubts. This finding basically confirms one the main outputs of the work by Bacchelli and Bird [21], who found that *code review is understandability*. The popularity of this category is similar in all the considered projects, confirming that this need is independent from the type of system or the developers working on it.

Still, a pretty popular need is 'Rationale' (N3). This has also to do with the understandability of the code change, however in this case it seems that a common reviewers' need

is having detailed information on the motivations leading the author to perform certain implementation choices.

Other categories are less diffused, possibly indicating that reviewers do not always need such types of information. For instance, 'Splittable' (N7) is the category having the lowest number of occurrences. This might be either because of the preventive operations that the development community adopts to limit the number of *tangled changes* [191] or because of the attention that developers put when performing code changes. In any case, this category seems to be less diffused and, as a consequence, one can claim that future research should spend more effort on different (most popular) reviewers' information needs.

RQ$_{1.3}$: *How do developers' perceive the identified needs?*

In this section, we present the results of our interviews and focus group with developers. First, we report on the participants' opinions on the taxonomy derived from the previous two sub-research questions, then we describe the most relevant themes that emerged from the analysis of the transcripts. We refer to individual interviewees using their identifiers ($P_{\#}$ and $FG_{\#}$).

Participants' opinion on the taxonomy. In general, all interviewees agreed on the information needs emerged from the code review threads: For all the categories, the developers agreed that they were asking those types of question themselves, several times and repeatedly. Furthermore, the order of importance of the categories was also generally agreed upon: According to the interviewees, the most important and discussed topic is 'suitability of an alternative solution' (N1), followed by 'understanding' (N2), 'rationale' (N3), and 'code context' (N4). Interestingly, the 'splittable' (N7) category is perceived as very important for the interviewed developers, but they confirmed that it happens rarely to receive big and long patches to review.

Although also the participants in the focus group agreed with the taxonomy of needs and their ranking, they stated that questions regarding 'correct understanding' (N2) are not common (in our taxonomy is ranked second). When discussing this difference with the focus group participants, they argued that this discrepancy was probably due to the type of projects we analyzed: Indeed, we analyzed open-source systems, while the focus group was conducted with participants working in an industrial, closed-source setting. One developer said: *"if I don't understand something of the change, I just go to my colleague that created it and ask to him. This is possible because we are all in the same office in the same working hours, while this is not the case in the projects you analyzed."*

Understanding a code change to review. An important step for all the interviewees when it comes to reviewing a patch is *to understand the rationale behind the changes* (N3). P_1 explained that to understand why the author wrote the patch, he first reads the commit message, since *"[it] should be enough to understand what's going on."* Interviewees said that it is very useful to have attached a ticket to the commit message, for example a JIRA issue, to really understand why it was necessary submitting the patch [P_{1-4}, FG_{1-3}]. However, sometime the patch is difficult to understand, and this leads to reviewers asking for more context or rationale of the change, as P_1 put it: *"Sometime the commit message just says "Yes, fix these things." And you say "Why? Was it broken? Is there a bug report information?" So in this case there is not enough description, and I would have to ask for it"*. Interestingly, P_1 reported that this issues generally happens with new contributors

or with novice developers. During the focus group, FG_3 said he also uses tests to obtain more context about the change: *"In general, to get more context I read the Java docs or the tests."* Finally, all the interviewees explained that to obtain more context, or the rationale behind the change, they use external IRC channels (outside Gerrit) to get in touch with reviewers/authors, *e.g.*, by emails, or Slack.

Authors' information needs. Considering the point of view of the *author* of a change, the interviewees explained that code review is sometimes used as a way to *get information* from specialized experts, thus underling the dual nature of the knowledge exchange happening in code review [169, 192]. P_2 explained that it is sometimes difficult for an author to have all the information they need to make the change, for example if the change is in a part of the system where they are not expert. In this case, P_2 explained: *"when you make a change, you usually add the experts of the system to your review, and then you ping them on IRC, asking for a review, if they have some time."* Interestingly, this point also came up during the focus group, where a developer said *"if it's a new system [...] my knowledge lacks at one front, it may be technology, it may be knowledge of the system."* In this case, the developer would ask the help of colleagues. This is also in line with another need we discovered in the previous research question, that is the 'Specialized Expertise' (N6). Indeed, interviewees said that when they are not familiar with the change, or they do not have the full context of the change, they ask an expert to contribute: *"[in the project where I work] we have sub-system maintainers: they are persons with knowledge in that area and have more pleasure or willingness to work on those specialized areas. If the reviewers do not reach consensus during the review, we always ask to those experts."* [P_4].

Small and concise patches. When discussing with developers the 'Splittable' (N7) need, all agreed that patches should be self-contained as much as possible [P_{1-4}, FG_{1-3}]. P_3 said: *"I always ask to split it, because in the end it will be faster to get it in [the system]."* P_4 added that it is something that they do all the time, because usually people do not see this issue. P_2 said: *"It's always better to have 10 small reviews than one big review with all the changes, because no one will review your code. It's like that. So if you want to merge something big, it's always better to do it in small changes."*

Another point raised during the interviews is that large patch sets are difficult to review and require a lot of time to read [P_{1-4}, FG_{1-3}], thus this may delay the acceptance of the patches. P_4 explained: *"You can have a large patch set that is 90% okay and 10% that not okay: the 10% will generate a lot of discussion and will block the merge of 90% of the code. So yes, it's something that I do all the time. I ask people, you need to organize better the patch."*

When talking about the issue, P_1 also added that having small patches is very important for making it easier to revert them: *"yes this is something I find it to be really, really important. Bugs are everywhere – there is always another bug to fix. So the patch should be small enough that [in case of bugs] you can revert it without breaking any particular code."*

Interestingly, all the interviewees agreed that tools could help reviewers and authors in solving this need: for example, when submitting a large patch, the tool could suggest the author to split it into more parts to ease the reviewing process.

Offering a solution. All interviewees agreed that to do a proper code review, reviewers should always pinpoint the weak parts in the code and offer a solution [P_{1-4}, FG_{1-3}]. P_3 said: *"When I request the change, I usually put a link or example because I know that maybe the other guy doesn't know about the other approach. This is usually the main reason*

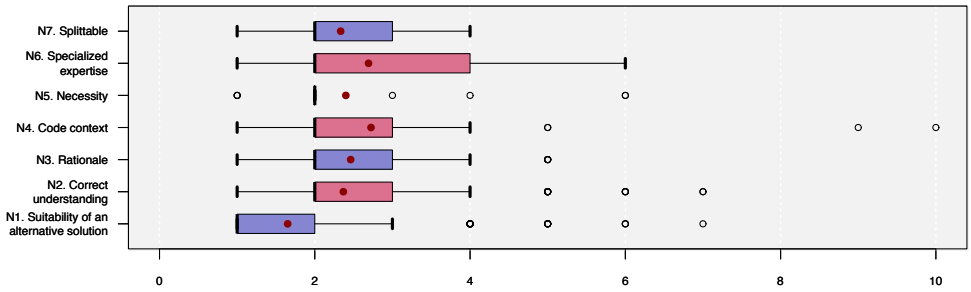


Figure 6.3: Distribution of the number of replies for each reviewers' need.

why somebody didn't do something: because he didn't know it was possible." P_1 added: *"[...] whenever you propose a change, you should always explain why you need to change it and what. Just putting [the score to] minus two, or even minus one without explanation, is bad because then people don't know what to do. We have to try being more friendly as a community."* This constructive behavior was also agreed upon during the focus group: one developer said that the worst thing that can happen in a code review is a non-constructive comment. Interestingly, this reported behavior confirms what we discovered in our previous research question: indeed, 'Suitability of an alternative solution' (N1) is the most frequent type of question when doing code review.

In addition, concerning constructive feedback, interviewees said that when they do not fully understand a change, they first ask the author explanations: *"For example, if you don't understand correctly the change this person is trying to add, you just ask him, and they are forced to answer you. And if you don't have the context information, they should be able to provide it to you."*[P_2] Interviewees said that it is better to ask explanation to the author first, and only after decide to/not merge the patch. P_4 also explained that sometime it is better to accept a patch than start a big discussion on small detail: *"Even though I understand that a better solution will be doable, I'll probably won't propose it because a lot of times people won't have time to actually rework on a new proposal, and you need to balance how you want the project to move forward: Sometimes it's better to have a code that is not the best solution, but at least does not regress and it fixes a bug."*

6.4.2 RQ_2 - The Role of Reviewers' Information Needs In A Code Review's Lifecycle

We present the results achieved when answering our second research question, which was focused on the understanding of the role of reviewers' information needs in the lifecycle of the code review process. We report the results by considering each sub-research question independently.

$RQ_{2.1}$: *What are the reviewers' information needs that attract more discussion?*

To answer **$RQ_{2.1}$** and understand to what extent the reviewers' needs attract developers' discussions, we compute, for each discussion thread that we manually categorized, the number of iterations that involve the developers of a certain code review.

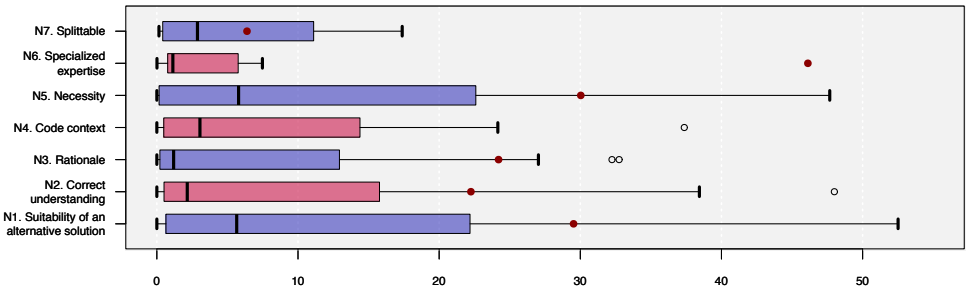


Figure 6.4: Distribution of the number of hours needed to answer each reviewers' need category.

Figure 6.3 depicts box plots reporting the distribution of the number of answers for each reviewers' need previously identified (red dots indicate the mean). Approximately 18% of code review threads (considering both merged and abandoned patches of every projects) do not have an answer. The first observation regards the median value of each distribution: as shown, all of them are within one and three, meaning that most of the threads are concluded with a small amount of discussion. From a practical perspective, this result highlights that authors can address almost immediately the need pointed out by a reviewer; at the same time, it might highlight that tools able to address the reviewers' needs identified can be particularly useful to even avoid the discussion and lead to an important gain in terms of time spent to review source code. Among the reviewers' needs, the 'Specialized expertise' (N6) is the one with the largest scattering of discussion rate. This result seems to indicate that the more collaboration is required due the largest number of replies a discussion receives, which possibly preclude the integration in the codebase of important changes that require the expertise of several people.

The statistical tests confirmed that there are no statistically significant differences among the investigated distributions, with the only exception of 'Suitability of an alternative solution' (N1), for which the ρ-value is lower than 0.01 and the Cliff's d is 'medium'. This category is the one having the lowest mean (1.7) and we observed that often authors of the code change tend to directly implement the alternative solution proposed by the reviewer without even answering to the original comment. This tendency possibly explain the motivation behind this statistical difference.

Overall, according to our results, most of the reviewers' information needs are satisfied with few replies–most discussions are closed shortly. The only category having more scattered results is the one where reviewers ask for the involvement of more people in the code review process.

RQ$_{2.2}$: *How long does it take to get a response to each reviewers' information need?*

Figure 6.4 reports the distribution of the number of hours needed to get an answer for each group of reviewers' information need. In this analysis we could only consider the questions having at least one answer; similarly, if a reviewer's comment got more than one reply, we considered only the first one to compute the number of hours needed to answer the comment.

Looking at the results, we can observe that the median is under 7 hours for almost all the categories. A possible reason for that consists of the nature of the development

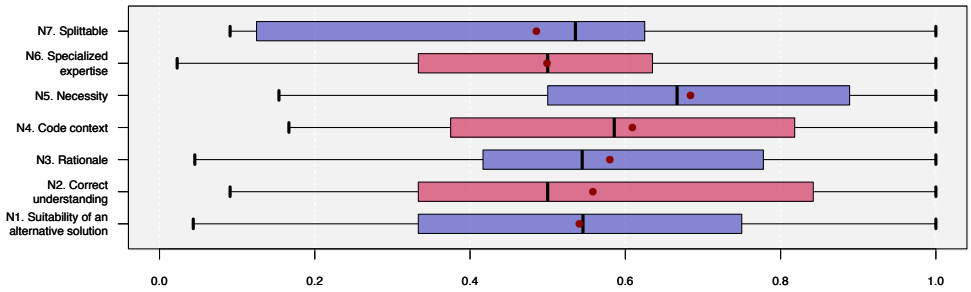


Figure 6.5: Distribution of reviewers' needs over different iterations of the code review process.

communities behind the subject systems. Indeed, all the projects have development teams that span across different countries and timezones: thus, we might consider as expected the fact to not have an immediate reaction to most of the comments made by reviewers. Some differences can be observed in the distributions of two reviewers' information needs such as 'Necessity' (N5) and 'Suitability of an alternative solution' (N1). In this case, the median number of hours is higher with respect to the other categories (7 vs 5), while the 3rd quartiles are around one day (meaning that 25% of the questions in this category took more than one day to have a response). Conversely, the discussion of other categories generally took less time to start. For instance, the 'Specialized expertise' (N6) need has a median of one hour and a 3rd quantile equal to four. Such differences, however, are not statistically significant.

To conclude the analysis of our findings for this research question, we can argue that developers generally tend to respond slower to questions regarding the proposal of alternatives and the evaluation of the actual necessity of a certain code change; on the other hand, questions where more reviewers are called to discuss seem to get a faster response time.

RQ$_{2.3}$: *How do the reviewers' information needs change over the code review process?*

The last research question targets the understanding of whether reviewers' needs vary over the different iterations of the code review process. Figure 6.5 presents the result of our analysis, with box plots depicting the distribution of each reviewers' information need in the various iterations: for the sake of better comprehensibility of the results, we considered the normalized number of iterations available in each of the 900 code review threads analyzed.[7]

Almost all the categories have their median around 0.5, meaning that the majority of reviewers' information needs are raised in the first half of the review process. Moreover, we are not able to map reviewers' information needs with any specific iteration. This result might indicate that there is not a time-sensitive relationship between those needs and that they arise independently from how much discussion has already been going on in the review.

Besides this general conclusion, we also notice some differences between the category 'Necessity' (N5) and the others. In the case of the 'Necessity' (N5) category the median and

[7]We also conducted an analysis using the absolute number of iterations, yet results were equivalent.

mean reach both 0.67, thus indicating that most of such questions come later in the process. It is interesting to note that modifications aimed at performing perfective changes that improve the overall design/style of the source code rather than solving issues are mainly requested by reviewers in a later stage of the code review process, *i.e.*, likely after that most important fixes solving problems impacting the functioning of the system are already submitted by the author and answers about understanding the context of the change are given. Such an observation may need further investigations and validation, however it may possibly reveal the possibility to devise strategies to guide the next generation of code review tools toward a *selection* of the information that a reviewer might need in a earlier/later stage of the code review process.

6.5 Threats to Validity

Our study might have been subject to a number of threats to validity that may affect our results. This section summarizes the limitations of our study and how we tried to mitigate them.

Validity of the defined reviewers' needs. Since the meaning of a question may be dependent by the context, we may lack of a full understanding of its nature and background. This type of threat may first apply to our study when we identify code review threads composed of both questions and answers: to this aim, we automatically mined the Gerrit repository that is a reliable source for the extraction of code review data [19, 193]. To extract code review threads we employed the publicly available APIs of such repository: For this reason, we are confident on the completeness of the extracted data.

The adopted open card sorting process is also inherently subjective because different themes are likely to emerge from independent card sorts conducted by the same or different people. To ensure the correctness and completeness of the categories associated to the reviewers' needs identified with the card sorting, we iteratively conducted the process by merging and splitting reviewers' need categories if needed. As an additional step, we also took into account authors responses and discussion threads when classifying questions made by reviewers, with the aim of properly understand the context in which a certain question has been made. Moreover, all the authors of this chapter, who have more than seven years of experience in software development, assessed the validity of the emerged categories, thus increasing its overall completeness. Of course, we cannot exclude the missing analysis of specific code review threads that point to categories that were not identified in our study.

We consider questions asked by reviewers through the Gerrit platform as indicators of the actual reviewers' needs. This assumption may not hold for all projects, as many active projects do not use the Gerrit platform. For example, Tsay *et al.* [194] highlighted how several developers contribute to the software development by using different platforms (*e.g.*, GitHub). However, we partly mitigated this threat to validity by carefully selecting software systems broadly studied in code review research [11, 32, 180] and having a large number of code review data (which indicates they actively use Gerrit). The study of different platforms such as GitHub, GitLab, or Collaborator is left for future work.

External validity. As for the generalizability of the results, we conducted this study

on a statistically significant sample of 900 code reviews that include more than 1,800 messages belonging to three well-known projects that use the Gerrit platform since 2011. A threat to validity in this category may arise when we consider closed-source projects. In that case, the experience of closed-source reviewers may affect the need to asks clarification questions, therefore, the findings that we found in open-source project may be not generalizable to a closed-source context. As part of our future research agenda, we plan to extend this study by including closed-source projects.

6.6 Discussion and Implications

Our quantitative and qualitative results showed that reviewers have a diversity of information needs at different conceptual levels and pertaining to different aspects of the code under review. In this section we discuss how our results lead to recommendations for practitioners and designers, as well as implications for future research.

Early detection of splittable changes. Even if the 'Splittable' (N7) category is the less frequently occurring, interviewees argued that it is really useful to automatically detect splittable code changes before a submission. For example, in the focus group all the participants (FG_{1-3}) suggested: *"if it's an unrelated change [...], pull it out of this ticket and put it on another issue."* In fact, this would (1) decrease the time spent in detecting this issue and asking the author to re-work the change, as well as (2) reduce the risks of introducing defects in the source code [191].

Researchers have already underlined the risks of tangled code changes (*i.e.*, non-cohesive code changes that may be divided in atomic commits) for mining software repositories approaches [191] and have proposed mechanisms for automatically splitting them. For instance, Herzig and Zeller [191] proposed an automated approach relying on static and dynamic analysis to identify which code changes should be separated; Yamauchi suggested a clustering algorithm tuned to identify unrelated changes in a commit message [195]; and Dias *et al.* [196] proposed a methodology to untangle code changes at a finer-granularity, *i.e.*, by selecting the single statement of a code review that should be placed in other commits. More recently researchers also proposed untangling techniques tailored explicitly to code review [190, 197] and conducted the first experiments to measure the effects of tangled code changes on code review [197, 198] substantiating the value of separating unrelated changes.

Despite these advances in splitting algorithms and their immediate practical value, no commercial code review tool offers this feature. Our analysis underlines even more the relevance of having such a feature integrated as early as possible in the development process, possibly in the development environment, so that authors send already self-contained patches for review. Moreover, despite the notable research advances in the field, we believe that there is still room for improvement, *e.g.*, by complementing state-of-the-art methods with conceptual-related information aimed at capturing the semantic relationships between different code changes. Also, early improvements of code changes before review are in line with the work by Balachandran [199]. He reported that the time to market can be reduced also creating automatic bots able to conduct preliminary reviews [199]. In this regard, there

are still plenty of opportunities and challenges on how and when bots can automatically help reviewers during their activities and whether they may be employed to assist some of the developers' needs in code review.

Synchronous communication support. The absence of a proper real-time communication channel within code review tools was a common issue that emerged from both the interviews with the open-source developers and the focus group. In fact, two interviewees ([FG_1] and [FG_3]) explained: *"you can just go to the author and ask to him in person, and maybe it would be a long discussion [...]"*. This is in line with the experience reported by developers at Microsoft in a previous study by Bacchelli and Bird [21]. Nevertheless, in-person discussions can happen only if both author and reviewer are co-located, otherwise logistic barriers could impose serious constraints [21]. Yet, open-source developers are able to fulfill this real-time communication need using alternative channels; P_2 stated: *"we usually have an IRC channel [...]"*. The two observations suggest that, when it is possible, developers prefer to rely on direct communication to discuss feedback; this may be to avoid discussing difficult criticism online in a public forum and to have a higher communication bandwidth than small online thread comments. In both scenarios, our results show that current code review tools are clearly not able to fully satisfy the communication need of the involved people. Future work should be conducted to understand how communication can be facilitated within the code review tool itself (thus improving traceability of discussions, which is relevant for future developers' information needs [171]); in principle, this future analysis should take into account not only technical aspects to increase the communication bandwidth, but also the social aspects that could currently hinder developers from discussing certain arguments with the current tools.

Automatic change summarization. 'Correct understanding' (N2) and 'Rationale' (N3) are also key information needs for reviewers. Normally this is achieved by perusing the code change description or additional comments. Nevertheless, our interviewees reported cases in which these sources of information were insufficient to fulfill this need; on this P_3 reported: *"I even had cases where the description didn't have anything in common with the code"*. Indeed this shows that another significant source of delay in a code review process is when patches contain unaligned or missing information (*i.e.*, the commit message is not clear enough or it does not match with the actual patch). Code summarization techniques appear to be a good fit for this task: Indeed, past literature presented different summarization techniques that can be used to both produce or check the current documentation. For example, Buse and Weimer proposed a technique to synthesize human-readable documentation starting from code changes [200], but also several other researchers have been contributing with more approaches: Canfora*et al.* experimented LDiff [201], Parnin and Görg developed CILDiff [202], and Cortés-Coy *et al.* designed ChangeScribe [203]. Our analysis suggests that supporting code review is a ripe opportunity for research on code summarization techniques to have another angle of impact on a real-world application.

6.7 Conclusions

Modern code review is an important technique used to improve software quality and promote collaboration and knowledge sharing within a development community. In a typical code review process, authors and reviewers interact with each other to exchange ideas, find bugs, and discuss alternative solutions to better design the structure of a submitted code change. Often reviewers are required to inspect author patches without knowing the rationale or without being aware of the context in which a code change is supposed to be plugged-in. Therefore, they must ask questions aimed at addressing their doubts, possibly waiting for a long time before getting the expected clarifications. This might potentially result in causing delays in the integration of important changes into production.

In this work we investigate the reviewers' information needs by analyzing 900 code review threads of three popular open-source software systems (OpenStack, Android, and QT). Moreover, we conduct four semi-structured interviews with developers from the considered projects and one focus group with developers from a software quality consultancy company, with the aim of challenging and discussing our outcome.

We discovered the existence of seven high-level reviewers' information needs, which are differently distributed and have, therefore, different relevance for reviewers. Furthermore, we analyzed the role played by each category of reviewers' information needs across the lifecycle of a code review, and in particular what are the reviewers' information needs that attract more discussion, for how long a reviewer should wait to get a response, and how the information needs change over the code review lifecycle.

Based on our findings, we provide recommendations for practitioners and researchers, as well as viable directions for impactful tools and future research. We hope that the insights we have discovered will lead to improved tools and validated practices which in turn may lead to higher code quality overall.

7

When Testing Meets Code Review: Why and How Developers Review Tests

Automated testing is considered an essential process for ensuring software quality. However, writing and maintaining high-quality test code is challenging and frequently considered of secondary importance. For production code, many open source and industrial software projects employ code review, a well-established software quality practice, but the question remains whether and how code review is also used for ensuring the quality of test code. The aim of this research is to answer this question and to increase our understanding of what developers think and do when it comes to reviewing test code. We conducted both quantitative and qualitative methods to analyze more than 300,000 code reviews, and interviewed 12 developers about how they review test files. This work resulted in an overview of current code reviewing practices, a set of identified obstacles limiting the review of test code, and a set of issues that developers would like to see improved in code review tools. The study reveals that reviewing test files is very different from reviewing production files, and that the navigation within the review itself is one of the main issues developers currently face. Based on our findings, we propose a series of recommendations and suggestions for the design of tools and future research.

Preprint: https://doi.org/10.5281/zenodo.1688846,

Data and materials: https://doi.org/10.5281/zenodo.4075318.

This chapter has been published as *Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review. In Proceedings of the 40th International Conference on Software Engineering (ICSE), New York, NY, USA, may 2018. ACM.*[20]

7.1 Introduction

Automated testing has become an essential process for improving the quality of software systems [70, 71]. Automated tests (hereafter referred to as just 'tests') can help ensure that production code is robust under many usage conditions and that code meets performance and security needs [70, 72]. Nevertheless, writing effective tests is as challenging as writing good production code. A tester has to ensure that test results are accurate, that all important execution paths are considered, and that the tests themselves do not introduce bottlenecks in the development pipeline [70]. Like production code, test code must also be maintained and evolved [56].

As testing has become more commonplace, some have considered that improving the quality of test code should help improve the quality of the associated production code [2, 204]. Unfortunately, there is evidence that test code is not always of high quality [56, 83]. Vazhabzadeh *et al.* showed that about half of the projects they studied had bugs in the test code [112]. Most of these bugs create false alarms that can waste developer time, while other bugs cause harmful defects in production code that can remain undetected. We also see that test code tends to grow over time, leading to bloat and technical debt [56].

As code review has been shown to improve the quality of source code in general [21, 205], one practice that is now common in many development projects is to use Modern Code Review (MCR) to improve the quality of test code. But how is test code reviewed? Is it reviewed as rigorously as production code, or is it reviewed at all? Are there specific issues that reviewers look for in test files? Does test code pose different reviewing challenges compared to the review of production code? Do some developers use techniques for reviewing test code that could be helpful to other developers?

To address these questions and find insights about test code review, we conducted a two-phase study to understand how test code is reviewed, to identify current practices and reveal the challenges faced during reviews, and to uncover needs for tools and features that can support the review of test code. To motivate the importance of our work, we first investigated whether being a test changes the chances of a file to be affected by defects. Having found no relationship between type of file and defects, then in the first phase, we analyzed more than 300,000 code reviews related to three open source projects (Eclipse, OpenStack and Qt) that employ extensive code review and automated testing. In the second phase, we interviewed developers from these projects and from a variety of other projects (from both open source and industry) to understand how they review test files and the challenges they face.

7.2 Background and Motivation

Past research has shown that both test code and production code suffer from quality issues [56, 83, 206]. We were inspired by the study by Vahabzadeh *et al.* [112] who showed that around half of all the projects they studied had bugs in the test code, and even though the vast majority of these test bugs were false alarms, they negatively affected the reliability of the entire test suite. They also found that other types of test bugs (*silent horrors*) may cause tests to miss important bugs in production code, creating a false sense of security. This study also highlighted how current bug detection tools are not tailored to detect test bugs [112], thus making the role of effective test code review even more critical.

Some researchers have examined MCR practices and outcomes and showed that code review can improve the quality of source code. For example, Bacchelli *et al.* [21] interviewed Microsoft developers and found that code reviews are important not only for finding defects or improving code, but also for transferring knowledge, and for creating a bigger and more transparent picture of the entire system. McIntosh *et al.* [180] found that both code review coverage and participation share a significant link with software quality, producing components with up to two and five additional post-release defects, respectively. Thongtanunam *et al.* [33] evaluated the impact that characteristics of MCR practices have on software quality, studying MCR practices in defective and clean source code files. Di Biase *et al.* [207] analyzed the Chromium system to understand the impact of MCR on finding security issues, showing that the majority of missed security flaws relate to language-specific issues. However, these studies, as well as most of the current literature on contemporary code review, either focus on production files only or do not explicitly differentiate production from test code.

Nevertheless, past literature has shown that test code is substantially different from production code. For instance, van Deursen *et al.* [2] showed that when refactoring test code, there is a unique set of code smells—distinct from that of production code—because improving test code involves additional test-specific refactoring. Moreover, test files have their own libraries that lead to specific coding practices: for example, Spadini *et al.* [137] studied a test practice called mocking, revealing that the usage of mocks is highly dependent on the responsibility and the architectural concern of the production class.

Furthermore, other literature shows that tests are constantly evolving together with production code. Zaidman *et al.* [56] investigated how test and production code co-evolve in both open source and industrial projects, and how test code needs to be adapted, cleaned and refactored together with production code.

Due to the substantial differences between test code and production code, we hypothesize that how they should be reviewed may also differ. However, even though code review is now widely adopted in both open source and industrial projects, how it is conducted on test files is unclear. We aim to understand how developers review test files, what developers discuss during review sessions, what tools or features developers need when reviewing test files, and what challenges they face.

7.3 Should Test Files Be Reviewed?

Even though previous literature has raised awareness on the prevalence of bugs in test files, such as the work by Vahabzadeh *et al.* [112], it may well be that these type of bugs constitute such a negligible number compared to defects found in production code that investing resources in reviewing them would not be advisable. If test code tends to be less affected by defects than production code, pragmatic developers should focus their limited time on reviewing production code, and research efforts should support this.

To motivate our research and to understand whether test code should be reviewed at all, we conducted a preliminary investigation to see whether test and production code files are equally associated with defects. To study this, we used a research method proposed by McIntosh *et al.* [180] where they analyzed whether code review coverage and participation had an influence on software quality. They built a statistical model using the post-release defect count as a dependent variable and metrics highly correlated with defect-proneness

as explanatory variables. They then evaluated the effects of each variable by analyzing their importance in the model [208].

We applied the same idea in our case study; however, as our aim was to understand whether test files are less likely to have bugs than production files, we added the type of file (*i.e.*, either test or production) as an additional explanatory variable. If a difference existed, we would expect the variable to be significant for the model. If the variable was not relevant, a test file should neither increase or decrease increase the likelihood of defects, indicating that test files should be reviewed with the same care as production files (assuming one also cares about defects with tests).

Similar to McIntosh *et al.* [180], we used the three most common families of metrics that are known to have a relationship with defect proneness as control variables, namely *product metrics* (size), *process metrics* (prior defects, churn, cumulative churn), and *human factors* (total number of authors, minor authors, major authors, author ownership). To determine whether a change fixed a defect (our dependent variable), we searched version-control commit messages for co-occurrences of defect identifiers with keywords like 'bug', 'fix', 'defect', or 'patch'. This approach has been used extensively to determine defect-fixing and defect-inducing changes [91, 92].

We considered the same systems that we used for the main parts of our study (Qt, Eclipse, and Openstack) as they perform considerable software testing and their repositories are available online. Due to the size of their code repositories, we analyzed a sample of sub-projects from each one of them. For QT and Openstack, we chose the five sub-projects that contained more code reviews: *qt*, *qt3d*, *qtbase*, *qtdeclarative*, *qtwebengine* (Qt), and *cinder*, *heat*, *neutron*, *nova*, and *tempest* (Openstack). Eclipse, on the other hand, does not use identifiers in commit messages. Therefore, we used the dataset provided by Lam *et al.* [209] for the *Platform* sub-project. This dataset includes all the bugs reported in the Eclipse bug tracker tool, together with the corresponding commit hash, files changed, and other useful information.

We measured dependent and independent variables during the six-month period prior to a release date in a release branch. We chose the release that gave us at least 18 months of information to analyze. In contrast to McIntosh *et al.*'s [180] work, we calculated metrics at the file level (not package level) to measure whether the file being test code vs. production code had any effect.

We observed that the number of buggy commits was much smaller compared to the number of non-buggy commits, *i.e.*, classes were imbalanced, which would bias the statistical model. Therefore, we applied SMOTE (Synthetic Minority Over-sampling Technique) [210] to make both classes balanced. All R scripts are available in our online appendix [50].

To rank the attributes, we used Weka [1], a suite of machine learning software written in Java. Weka provides different algorithms for identifying the most predictive attributes in a dataset—we chose *Information Gain Attribute Evaluation (InfoGainAttributeEval)*, which has been extensively used in previous literature [211–213]. InfoGainAttributeEval is a method that evaluates the worth of an attribute by measuring the information gain with respect to the class. It produces a value from 0 to 1, where a higher value indicates a stronger influence.

[1] https://www.cs.waikato.ac.nz/ml/weka/

The precision and recall of the resulting model were above 90%, indicating that it is able to correctly classify whether most of the files contain defects, strengthening the reliability of the results.

Table 7.1: Ranking of the attributes, by decreasing importance

Attribute	Average merit	Average Rank
Churn	0.753 ± 0.010	1 ± 0
Author ownership	0.599 ± 0.002	2.2 ± 0.4
Cumulative churn	0.588 ± 0.013	2.8 ± 0.4
Total authors	0.521 ± 0.001	4 ± 0
Major authors	0.506 ± 0.001	5 ± 0
Size	0.411 ± 0.027	6 ± 0
Prior defects	0.293 ± 0.001	7 ± 0
Minor authors	0.149 ± 0.001	8 ± 0
Is test	0.085 ± 0.001	9 ± 0

We ran the algorithm using 10-fold cross-validation. Table 7.1 shows the results of the importance of each variable in the model as evaluated by *InfoGainAttributeEval*. The variable *is test* was consistently ranked as the least important attribute in the model, while *Churn*, *Author ownership*, and *Cumulative churn* were the most important attributes (in that order) for predicting whether a file will likely contain a bug. This is in line with previous literature.

From this preliminary analysis, we found that the decision to review a file should not be based on whether the file contains production or test code, as this has no association with defects. Motivated by this result, we conducted our investigation of practices, discussions, and challenges when reviewing tests.

7.4 Research Methodology

The main *goal* of our study is to increase our understanding of how test code is reviewed. To that end, we conducted *mixed methods research* [214] to address the following research questions:

RQ_1: How rigorously is test code reviewed? Previous literature has shown that code changes reviewed by more developers are less prone to future defects [205, 215], and that longer discussions between reviewers help find more defects and lead to better solutions [180, 194]. Based on our preliminary study (Section 7.3) that showed how the type of file (test vs. production) does not change its chances of being prone to future defects, and to investigate the amount of effort developers expend reviewing test code, we measured how often developers comment on test files, the length of these discussions, and how many reviewers check a test file before merging it.

RQ_2: What do reviewers discuss in test code reviews? In line with Bacchelli & Bird [21], we aimed to understand the concerns that reviewers raise when inspecting test files. Bacchelli & Bird noted that developers discuss possible defects or code improvements, and share comments that help one understand the code or simply acknowl-

edge what other reviewers have shared. We investigated if similar or new categories of outcomes emerge when reviewing test code compared with production code to gather evidence on the key aspects of test file reviews and on the reviewers' needs when working with this type of artifact.

RQ_3: Which practices do reviewers follow for test files? Little is known about developer practices when reviewing test files. To identify them, we interviewed developers from the 3 open source projects analyzed in the first two RQs, as well as developers from other projects (including closed projects in industry). We asked them how they review test files and if their practices are different to those they use when reviewing production files. This helped discover review patterns that may guide other reviewers and triangulate reviewers' needs.

RQ_4: What problems and challenges do developers face when reviewing tests? We elicited insights from our interviews to highlight important issues that both researchers and practitioners can focus on to improve how test code is reviewed.

In the following subsections, we discuss the three data collection methods used in this research. Section 7.4.1 describes the three open source projects we studied and Section 7.4.2 explains how we extracted quantitative data related to the prevalence of code reviews in test files. Section 7.4.3 discusses the manual content analysis we conducted on a statistically significant data sample of comments pertaining to reviews of test code. Section 7.4.4 describes the interview procedure used to collect data about practices, challenges, and needs of practitioners when reviewing test files.

7.4.1 Project Selection

To investigate what the current practices in reviewing test files are, we aimed at choosing projects that (1) test their code, (2) intensively review their code, and (3) use Gerrit, a modern code review tool that facilitates a traceable code review process for git-based projects [180].

The three projects we studied in our preliminary analysis (discussed in Section 3), match these criteria: **Eclipse**, **Openstack** and **Qt** and we continue to study these projects to answer our research questions. Moreover, these projects are commonly studied in code review research [33, 180, 193]. Table 7.2 lists their descriptive statistics.

Table 7.2: Subject systems' details after data pre-processing

	# of prod. files	# of test files	# of code reviews	# of reviewers	# of comments
Eclipse	215,318	19,977	60,023	1,530	95,973
Openstack	75,459	48,676	199,251	9,221	894,762
Qt	159,894	8,871	114,797	1,992	19,675
Total	450,671	77,524	374,071	12,743	1,010,410

7.4.2 Data Extraction and Analysis

To investigate how developers review test files, we extracted code review data from the Gerrit review databases of the systems under study. Gerrit explicitly links commits in a Version Control System (VCS) to their respective code review. We used this link to connect commits to their relevant code review, obtaining information regarding which files have been modified, the modified lines, and the number of reviewers and comments in the review.

Each review in Gerrit is uniquely identified by a hash code called Change-ID. After a patch is accepted by all the reviewers, it is automatically integrated into the VCS. For traceability purposes, the commit message of the integrated patch contains the Change-ID; we extracted this Change-ID from commit messages to link patches in the VCS with the associated code review in Gerrit.

To obtain code review data, we created GerritMiner, a tool that retrieves *all* the code reviews from Gerrit for each project using the Gerrit REST API [2]. The tool saves all review-related data (*e.g.*, Change-ID, author, files, comments, and reviewers) in a MySQL database. Through GerritMiner, we retrieved a total of 654,570 code reviews pertaining to the three systems. Since we were interested in just production and test files, we only stored code reviews that changed *source code* files (*e.g.*, we did not consider '.txt' file, 'README', JSON files, configuration files). After this process, we were left with 374,071 reviews. Table 7.2 presents the statistics.

To answer RQ_1, we selected only reviews that contained less than 50 files and had at least one reviewer involved who was different than the author [216–218]. In fact, as explained by Rigby *et al.* [219], a code review should ideally be performed on changes that are small, independent, and complete: a small change lets reviewers focus on the entire change and maintain an overall picture of how it fits into the system. As we were interested in code reviews where reviewers actually examined the code closely, we did not consider reviews where the author was the only reviewer. We also did not consider bots as reviewers (*e.g.*, Jenkins and Sanity Bots). At the end, the distribution of the number of reviewers per review (excluding the author) is the following: 27% have 1 reviewer, 26% have 2, 16% have 3, 10% have 4, and 20% have more than 4.

To understand how often and extensively discussions are held during reviews about test files, we considered the following metrics as proxies, which have been validated in previous literature [28]: number of comments in the file, number of files with comments, number of different reviewers, and the length of the comments. We only considered code reviews that contained at least one comment because we were interested in understanding whether there was any difference between review discussions of test and production files, and reviews that do not contain any discussion are not useful for this investigation.

We used the production file metrics as a baseline and separately analyzed the three different review scenarios based on what needed to be reviewed: (1) both production and test files, (2) only production files, or (3) only test files.

7.4.3 Manual Content Analysis

To answer RQ_2, we focused on the previously extracted comments (Section 7.4.2) by practitioners reviewing test files. To analyze the content of these comments, we performed a

[2] https://gerrit-review.googlesource.com/Documentation/rest-api.html

manual analysis similar to Bacchelli & Bird [21]. Due to the size of the total number of comments (1,010,410), we analyzed a statistically significant random sample. Our sample of 600 comments was created with a confidence level of 99% and error (E) of 5% [220].

7.4.4 Interviews

To answer RQs 3 and 4, guided by the results of the previous RQs, we designed an interview in which the goal was to understand *which practices* developers apply when reviewing test files. The interviews were conducted by the first author of this chapter and were semi-structured, a form of interview often used in exploratory investigations to understand phenomena and seek new insights [221].

Each interview started with general questions about code reviews, with the aim of understanding why the interviewee performs code reviews, whether they consider it an important practice, and how they perform them. Our interview protocol also contained many questions derived from the results of previous research questions. Our full interview protocol is available in the appendix [50]. We asked interviewees the following main questions:

1. What is the importance of reviewing these files?
2. How do you conduct reviews? Do you have specific practices?
3. What are the differences between reviewing test files and production files?
4. What challenges do you face when reviewing test files? What are your needs related to this activity?

During each interview, the researcher summarized the answers, and before finalizing the meeting, these summaries were presented to the interviewee to validate our interpretation of their opinions. We conducted all interviews via Skype. With the participants' consent, the interviews were recorded and transcribed for analysis. We analyzed the interviews by initially assigning codes [152] to all relevant pieces of information, and then grouped these codes into higher-level categories. These categories formed the topics we discuss in our results (Section 7.5).

We conducted 12 interviews (each lasting between 20 and 50 minutes) with developers that perform code reviews as part of their daily activities. Three of these developers worked on the projects we studied in the previous RQs. In addition, we had one participant from another open source project and 8 participants from industry. Table 7.3 summarizes the interviewees' demographics.

7.4.5 Threats to Validity and Limitations

We describe the threats to validity and limitations to the results of our work, as posed by the research methodology that we applied.

Construct validity. When building our model we assume that each post-release defect has the same importance, when in reality this could not be the case. We mitigate this issue analyzing only the release branch of the systems, which are more controlled than a development branch, to ensure that only the appropriate changes will appear in the upcoming release [180].

Table 7.3: Interviewees' experience (in years) and working context (OSS project, or company)

ID	Years as developer	Years as reviewer	Working context
P1	5	5	OSS
P2	10	10	Eclipse
P3	10	10	Company A
P4	20	6	Qt
P5	13	8	Company B
P6	5	5	Openstack
P7	7	5	Company C
P8	5	1	Company D
P9	11	3	Company E
P10	9	9	Company F
P11	7	2.5	Company D
P12	6	4.5	Company D

The content analysis was performed manually, thus giving rise to potentially subjective judgement. To mitigate this threat, we employ the negotiated agreement technique [221] between the first and second authors until agreement was reached (after 60 comments).

Internal validity – Credibility. Threats to *internal validity* concern factors we did not consider that could affect the variables and the relations being investigated. In our study, we interview developers from the studied software to understand how they review test files. Every developer has a specific way of reviewing, which may differ from the practices of other practitioners. We try to mitigate this issue by interviewing a range of developers from different open-source and industry projects. In addition, their interviewees' opinions may also be influenced by other factors, such as current literature on MCR, which could may have led them to social desirability bias [222], or by practices in other projects that they participate in. To mitigate this issue, we constantly reminded interviewees that we were discussing the code review practices specifically of their project. At the end of the interview, we asked them to freely talk about their ideas on code reviews in general.

Generalizability – Transferability. Our sample contains three open-source systems, which is small compared to the overall population of software systems that make use of code reviews. We reduce this issue by considering diverse systems and by collecting opinions from other open-source projects as well as from industry.

7.5 Results

In this section, we present the results to our research questions that aimed to understand how rigorously developers review tests, what developers discuss with each other in their reviews of test code, and what practices and challenges developers use and experience while performing these code reviews.

Table 7.4: The prevalence of reviews in test files vs production files (baseline)

Code review	# of files	# of files w/ comments	# of files wo/ comments	odds ratio	# of comments	Avg number of comments	Avg # of reviewers	Avg length of comments
Production	157,507	68,338 (43%)	89,169 (57%)	1.90	472,020	3.00	5.49	19.09
Test	102,266	29,327 (29%)	72,939 (71%)		129,538	1.27		15.32
Only production	74,602	32,875 (44%)	41,725 (56%)	0.87	122,316	1.64	3.95	18.13
Only test	22,732	10,808 (48%)	11,924 (52%)	1.15	52,370	2.30	5.15	17.01

RQ_1. How rigorously is test code reviewed?

In Table 7.4, we show the distribution of comments in code reviews for both production and test files when they are in the same review, and for code reviews that only contain either type.

Discussion in code reviews of test files. The number of test file reviews that contain at least one comment ranges from 29% (in reviews that combine test and production files) to 48% (in reviews that only look at test files). The number of production files that contain at least a single comment ranges from 43% (when together with test files) to 44% (in reviews that only look at production files).

In a code review that contains both types of files, the odds of a production file receiving a comment is 1.90 [1.87 – 1.93] higher than with test files. On the other hand, when a review only contains one type of file, the odds of a test file receiving a comment is higher than that of a production file: 1.15 [1.11 – 1.18].

We also observed a large number of files that did not receive any discussion. The number of code files that did not receive at least a single comment ranges from 52% (in reviews that only look at test files) to 71% (in reviews that combine test and production files).

7

Discussion intensity in test files. In the code reviews that contain both types of files, production files received more individual comments than test files (3.00 comments per file for production, 1.27 comments for tests). The difference is statistically significant but small (Wilcoxon p-value< 2.2^{e-16}, Cliff's Delta=-0.1643); this is due to the large number of files with no comments (median = 0 in both test and production, 3_{rd} quantile = 1). The difference is larger when we analyze only files with at least a single comment (Wilcoxon p-value< 2.2^{e-16}, Cliff's Delta=-0.1385).

Again, numbers change when both files are not bundled in the same review. Code reviews on only production files contain fewer individual comments on average than reviews on only test files (1.64 comments for production, 2.30 comments for tests). The difference is statistically significant but small (Wilcoxon p-value< 2.2^{e-16}, Cliff's Delta=0.0585).

Production files receive longer comments than test files on average, both when they are in the same review (an average of 19.08 characters per comment in a production file against 15.32 in a test) and when they are not (18.13 against 17.01). The difference is again statistically significant but small (Wilcoxon p-value< 2.2^{e-16}, Cliff's Delta=0.0888).

Reviewers of test files. The number of reviewers involved in reviews containing both files and only tests is slightly higher compared to reviews containing production files. However, from the Wilcoxon rank sum test and the effect size, we observe that the overall difference is statistically significant but small (Wilcoxon p-value< 2.2^{e-16}, Cliff's Delta=−0.1724).

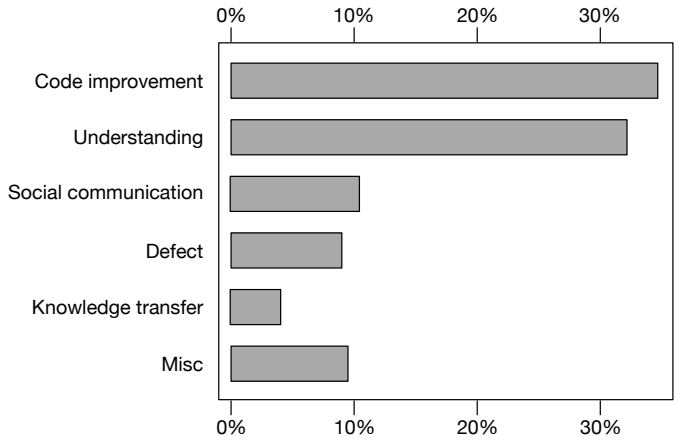


Figure 7.1: The outcomes of comments in code review of test files, after manual analysis in 600 comments (CL=99%, CI=5).

Test files are almost 2 times less likely to be discussed during code review when reviewed together with production files. Yet, the difference is small in terms of the number and length of the comments, and the number of reviewers involved.

RQ_2. What do reviewers discuss when reviewing test code?

In Figure 7.1, we report the results of our manual classification of 600 comments. When compared to the study by Bacchelli & Bird [21] that classified production and test code together, we exclude the 'Testing' category as all our comments were related to test code. In addition, we did not observe any comments related to 'External impact' and 'Review tool'. Interestingly, the magnitude of the remaining outcomes is the same as found by Bacchelli & Bird [21].

Code improvements (35%**).** This is the most frequently discussed topic by reviewers when inspecting test files. This category includes suggestions to use better coding practices, fix typos, write better Java-docs, and improve code readability. Interestingly, the code improvements that we found are also similar to the ones found by Bacchelli & Bird [21]. Yet the reviewers mostly discuss improvements focused on testing, as opposed to generic code quality practices, such as maintainability, which are the focus of reviews on production code [19, 21].

More specifically, 40% of the code improvement comments concern improvements to testing practices, such as better mocking usage and test cohesion. In addition, we found that in 12% of the cases, developers were discussing better naming for test classes, test methods, and variables. Interestingly, our interviewees mentioned that naming is important when reviewing test files, as P9 put it: "Most of the time I complain of code styling. Sometimes it's difficult to understand the name of the variables, so I often complain to change the names."

Of the code improvement comments, 14% are about untested and tested paths. According to our interviewees, this is an important concern when reviewing test files. Some examples of such review discussion comments are "Where is the assert for the non synchronized case?", "Add a test for context path of /.", and "I wouldn't do this test. That's an implementation detail."

Another 6% of the comments concern wrong assertions. As we discuss in the following RQs, developers often complain about readability of the assertions, namely the assertion has to be as specific as possible to let the developer better understand why the test failed. As an example, a developer asked to change an *assertTrue(equals())* to an *assertEqual()* in one comment.

Finally, we observed that 12% of the comments concern unused or unnecessary code, and 17% of them mention code styling. These kinds of comments are in line with those found on reviews of production code [19, 21].

Understanding (32%). This category represents all the comments where reviewers ask questions to better understand the code, including posing questions asking for explanations, suggestions, and motivating examples. In this category, we included comments such as "why do you need this for?", "what does this variable name mean?" and "why is this class static?". Interestingly, as opposed to review comments related to code improvements, the comments in this category did not reveal any differences from what we found in the test and production files analyzed in our previous work, *i.e.*, there were no comments that were specifically related to testing practices, such as assertion and mocking. This provides additional evidence on the importance of understanding when performing code reviews [21], regardless of the types of files under review.

Defect finding (9%). Within this category, we see discussion concerning test defects such as wrong assert handling, missing tests, misuse of test conventions, and incorrect use of mocks. We observe three different categories of defects: severe, not severe, and wrong assertions. More specifically, 43% of the comments are about severe issues, *i.e.*, tests that completely fail because of a wrong variable initialization, a wrong file path, or incorrect use of mocks. On the other hand, 41% of the comments are focused on less severe issues, such as missing test configurations. Finally, 14% of the comments concern wrong assertion handling, such as assertions of wrong scenarios. Interestingly, as opposed to the results reported by Bacchelli & Bird who found that "review comments about defects … mostly address 'micro' level and superficial concerns" [21], a large portion of the defects discussed in test file code reviews concern rather high-level and severe issues. A hypothesis for this difference may be that a good part of the severe, high-level defects that affect test files are localized (*i.e.*, visible just looking at the changed lines), while production files are affected by more delocalized defects that may be harder to detect by simply inspecting the changed lines.

Knowledge transfer (4%). This category, which also emerged in previous work [21], represents all the comments where the reviewers direct the committer of the code change to an external resource (*e.g.*, internal documentation or websites). We observe two different types of comments in this category: comments that link to external resources and that contain examples. More specifically, 55% of these comments contain links to external documentation (*e.g.*, Mockito website, python documentation), to other classes of the project (*e.g.*, other tests), and to other patches. The rest of the comments are examples where the

reviewer showed how to tackle the issue with an example, within the review comment itself, of how s/he would do it.

Social communication (11%). Finally, this category, in line with the work by Bacchelli & Bird [21], includes all the comments that are social in nature and not about the code, examples such as "Great suggestion!" and "Thank you for your help".

Reviewers discuss better testing practices, tested and untested paths, and assertions. Regarding defects, half of the comments regard severe, high-level testing issues, as opposed to results reported in previous work, where most of the comments on production code regarded low level concerns.

RQ_3. Which practices do reviewers follow for test files?

We analyze the answers obtained during our interviews with developers. We refer to individual interviewees using ($P_{\#}$).

Test driven reviews. Regardless of having test files in the patch, all participants agreed that before diving into the source code, they first get an idea of what the change is about, by reading the commit message or any documentation attached to the review request [$P_{1,2,6-10,12}$]. P_2 added: "I look at what it says and I think to myself 'how would I implement that?', just to give a sense of what are the files that I would be expecting to be changed, what are the areas of the system that I'm expecting that they touch." If the files that are attached to the patch look much different from what they expected, they immediately reject the code change [$P_{2,7,9,10}$].

Once they understood what the change is about, developers start to look at the code. In this case, we identified two different approaches: some developers prefer to *read test files first* followed by production files [$P_{2,4,5}$], while other developers prefer to *read production files first* and then review tests [$P_{1,3,6-10}$].

When starting from tests, developers say they can understand what the feature should do even before looking at its implementation [$P_{2,4,5}$]. P_5 says: "It is similar to read interfaces before implementations. I start from the test files because I want to understand the API first." In this approach, developers first see what the code is tested for, then check whether the production code does only what is tested for or if it does more than what is necessary [$P_{2,4,5}$]: "If I start to find something in production that is much different of what I am inferring from the tests, those are my first questions" [P_5].

More developers instead start reviewing the change in its production files first [$P_{1,3,6-10}$]. As P_8 explained: "I start first from the production code, because I can have a sense of what should be tested." The advantage of such an approach is that developers do not waste time validating whether the test covers every path for a change that is wrong in first place [$P_{1,3,6-10}$]. P_7 also said that he would prefer to start reviewing the tests, but the poor quality of the tests makes this not realistic: "I would prefer to starting with the tests, the problem is that tests are usually very bad. And the reason is because they usually tend to test what they implemented and not what they should have implemented. In TDD I would also start to review tests first but, as you know, this is not the case."

Similarly to when writing new code, when reviewing some developers prefer to start from tests, others from production. Reviewers who start inspecting tests use them to determine what the production code should do and whether it does only that. Reviewers who start inspecting production code prefer to understand the logic of production code before validating whether its tests cover every path.

Reviewers look for different problems when reviewing tests. According to the interviewees, reviewing test files require different practices to those used for reviewing production files; this is in line with the differences we found in the content of the subcategories of comments left for test files vs. production files for RQ2. Interviewees explain that it is especially different in terms of *what* to look for. P_{10} said that when reviewing production code they discuss more about the design and cleanliness of the code, whereas for reviews of tests they discuss more on tested/untested paths, testing practices like mocking and the complexity of the testing code.

A main concern for all the developers is understanding if all the possible paths of the production code are tested, especially corner cases [P_{8-12}]. "If the method being tested receives two variables, and with these two variables it can have on average five to ten different returns, I'll try to see whether they cover a sufficient number of cases so that I can prove that that method won't fail."[P_{11}]

Interviewees explained that they often complain about maintainability and readability of the test (*i.e.*, complex or duplicated code, if the test can be simpler, or divided into two smaller tests), but especially the name of the tests and the assertions [P_{7-12}]. "I often complain on the assertions, some assertions are really difficult to understand, we should try to be as specific as possible."[P_8]

A main concern of reviewers is understanding whether the test covers all the paths of the production code and to ensure tests' maintainability and readability.

Having the contextual information about the test. As we will discuss in the next RQ, a main concern for developers is the small amount of information provided in the code review [$P_{1,6,8-10,12}$]. Indeed, within a code review, reviewers can only see files that are changed and only the lines that have been modified, while interviewees complain that they do not have the possibility to, for example, automatically switch between production code and its test code, or to check other related test cases that are not modified in the patch [$P_{1,8,9}$].

For this reason, two developers explained that they check out the code under review and open it with another tool, for example a local IDE [$P_{9,12}$]. In this way, they can have the full picture of what is changed and get full support of other tools: "I particularly like opening the pull request to know what has come in again, which classes have been edited. I [open] GitHub, I access the PR and open it [in] my IDE. So I can look at the code as a whole, not just the files changed as it is in GitHub."[P_{12}] Another advantage of checking out the commit with the patch is that developers can see the tests running: "Most of the time I pull the pull request to see the feature and tests running, so I can have a sense of what have changed and how."[P_9] Nevertheless, this practice can only be accomplished

when the code base is limited in size and the coding environment can be easily pulled to the local machine of the reviewers.

> *Due to the lack on test-specific information within the code review tool, we observed that developers check out the code under review and open it in a local IDE: This allows them to navigate through the dependencies, have a full picture of the code, and run the test code. However this workaround is limited to small scale code bases.*

RQ_4. What problems and challenges do developers face when reviewing tests?

Test code is substantially different than production code. According to our interviewees, even if sometimes writing tests is simpler than writing production code [$P_{1,3,4,6-9,11}$], this changes when reviewing tests [$P_{3,4,6-9}$]: "Imagine you have to test a method that does an arithmetic sum. The production code only adds two numbers, while the test will have to do a positive test, a negative and have about fifteen different alternatives, for the same function."[P_{11}]

According to our interviewees, reviewing test files becomes complicated due to lack of context [$P_{1,6,8-10,12}$]. When reviewing a test, code review tools do not allow developers to have production and test files side-by-side [P_{12}]: "Having the complete context of the test is difficult, we often use variables that are not initialized in the reviewed method, so often I have to go around and understand where the variable is initialized."[P_8] Another developer said that "It's hard to understand which test is actually testing this method, or this path in the function."[P_1] and that when the test involves a lot of other classes (it is highly coupled) s/he never knows whether and how the other classes are tested [P_9].

Furthermore, one of the main challenges experienced by our interviewees is that often reviewing a test means reviewing code additions, which is more complicated than reviewing just code changes [$P_{2,11,12}$]. "Code changes make me think, why is this line changing from the greater than side to a less than side? While code additions you have to think what's going on in there, and tests are almost all the time new additions."[P_2] According to developers, test code is theoretically written once and if it is written correctly it will not change [P_2]. The reason is that while the implementation of the feature may change (*e.g.*, how it is implemented), both the result and the tests will stay the same [P_{11}].

> *Reviewing test files requires developers to have context about not only the test, but also the production file under test. In addition, test files are often long and are often new additions, which makes the review harder to do.*

The average developer believes test code is less important. "I will get a call from my manager if there is a bug in production while I'm not going to get a call if there's a bug in the test right?"[P_2] According to our interviewees, developers choose saving time to the detriment of quality. This is due to the fact that there is no *immediate* value on having software well tested; as P_7 explained "it is only good for the long term." For a product that is customer-driven, it is more important to release the feature on time without bugs,

because that is the code that will run on the client's machine [$P_{2,4-7,9}$]. P_7 said: "If I want to get a good bonus by the end of the year I will make sure that my features make it into production level code. If instead we would start to get punished for bugs or bad code practices, you will see a very very different approach, you would see way more discussions about tests than production code." Interviewees affirmed that the main problem is the developers mindset [$P_{1,2,7,8}$]: "It is the same reason as why people write bad tests, testing is considered as secondary class task, it is considered not but it is not."[P_7] Developers see test files as less important, because a bug in a test is a developer's problem while a bug in production code is a client's problem. As explained by P_7, developers are not rewarded for writing good code, but for delivering features the clients want.

Furthermore, according to our interviewees, during a review sometimes they do not even *look* at the test files, their presence is enough [$P_{1,3-6}$]. As P_6 said "Sometimes you don't look at the test because you see there is the file, you know that the code is doing what it has to do and you trust the developer who wrote it (maybe we trust too much sometimes)."

Developers have a limited amount of time to spend on reviewing and are driven by management policies to review production code instead of test code which is considered less important.

Better education on software testing and reviewing. Most interviewees agreed on the need to convince developers and managers that reviewing and testing are highly important for the software system overall quality [$P_{1,2,4,6-8,10}$]. Educating developers on good and bad practices and the dangers of bad testing [$P_{2,7,10}$]. "I would love to see in university people teaching good practice on testing. Furthermore, people coming from university they have no freaking clue on how a code review is done. Educating on testing and reviewing, how to write a good test and review it."[P_7]

Furthermore, with the help of researchers, developers could solve part of the education problem: one interviewee said that research should focus more on developers' needs, so that tool designers can take advantage of these needs and improve their tools [P_6]. "I think it is important to give this feedback to the people who write [code review] tools so they can provide the features that the community wants. Having someone like you in the middle, collecting this feedback and sending them to tool developers, this could be very helpful."[P_6]

Novice developers and managers are not aware of what is the impact of poor testing and reviewing on software quality, education systems should fix this. Moreover, research should focus more on developers' needs and expose them to tool makers to have an impact.

Tool improvements. Part of the interview was focused on what can be improved in the current code review tools.

According to our interviewees, the navigation between the production and the test files within the review is difficult [$P_{2,9,10,12}$]. "We don't have a tool to easily switch between

your tests and your production code, we have to go back and forth, then you have to look for the same name and trying to match them."[P_2] As mentioned before, the context of the review is limited to the files attached to the review itself, and this makes it difficult to have a big picture of the change. For example, test files are usually highly coupled to several production classes: however, developers can not navigate to the dependencies of the test, or other test in general, without opening a new window [$P_{2,9}$]. "If we could click on the definition of a class and go to its implementation would be amazing. That's why I pull the PR every-time and I lose a lot of time doing it."[P9] P12' said: "It's very unproductive to review in GitHub, because you first visualize all the codes, and then at the end are all the tests, and it ends up being more difficult having to keep going in the browser several times."

In addition, adding fine-grained information about code coverage during the review is considered helpful [$P_{1,7,11,12}$]. More specifically, which tests cover a specific line [$P_{1,7,11,12}$], what paths are already covered by the test suite [$P_{2,12}$], and whether tests exercise exceptional cases [P_{12}]. Regarding the latter, P_{12} says: "I think it's harder to automate, but it is to ensure that not only the "happy" paths are covered. It is to ensure that the cover is in the happy case, in case of errors and possible variations. A lot of people end up covering very little or too much."

Tool features that are not related to test also emerged during our interviewees. For example, enabling developers to identify the importance of each file within the code review [$P_{7,8,11,12}$] and splitting the code review among different reviewers [P_7].

Review tools should provide better navigation between test and production files, as well as in-depth code coverage information.

7.6 Conclusions

Automated testing is nowadays considered to be an essential process for improving the quality of software systems. Unfortunately, past literature showed that test code, similarly to production code, can often be of low quality and may be prone to contain defects [112]. To maintain a high code quality standard, many software projects employ code review, but is test code typically reviewed and if so, how rigorously? In this chapter we investigated whether and how developers employ code review for test files. To that end, we studied three OSS projects, analyzing more than 300,000 reviews and interviewing three of their developers. In addition, we interviewed another 9 developers, both from OSS projects and industry, obtaining more insights on how code review is conducted on tests. Our results provide new insights on what developers look for when reviewing tests, what practices they follow, and the specific challenges they face.

In particular, after having verified that a code file that is a test does not make it less likely to have defects—thus little justification for lower quality reviews—we show that developers tend to discuss test files significantly less than production files. The main reported cause is that reviewers see testing as a secondary task and they are not aware of the risk of poor testing or bad reviewing. We discovered that when inspecting test files, reviewers often discuss better testing practices, tested and untested paths, and assertions.

Regarding defects, often reviewers discuss severe, high-level testing issues, as opposed to results reported in previous work [21], where most of the comments on production code regarded low level concerns.

Among the various review practices on tests, we found two approaches when a review involves test and production code together: some developers prefer to start from tests, others from production. In the first case, developers use tests to determine what the production code should do and whether it does only that, on the other hand when starting from production they want to understand the logic before validating whether its tests cover every path. As for challenges, developers' main problems are: understanding whether the test covers all the paths of the production code, ensuring maintainability and readability of the test code, gaining context for the test under review, and difficulty reviewing large code additions involving test code.

Finally, we provide recommendations for practitioners and educators, as well as viable directions for impactful tools and future research.

8

Test-Driven Code Review: An Empirical Study

Test-Driven Code Review (TDR) is a code review practice in which a reviewer inspects a patch by examining the changed test code before *the changed production code. Although this practice has been mentioned positively by practitioners in informal literature and interviews, there is no systematic knowledge of its effects, prevalence, problems, and advantages.*

In this chapter, we aim at empirically understanding whether this practice has an effect on code review effectiveness and how developers' perceive TDR. We conduct (i) a controlled experiment with 93 developers that perform more than 150 reviews, and (ii) 9 semi-structured interviews and a survey with 103 respondents to gather information on how TDR is perceived. Key results from the experiment show that developers adopting TDR find the same proportion of defects in production code, but more in test code, at the expenses of fewer maintainability issues in production code. Furthermore, we found that most developers prefer to review production code as they deem it more critical and tests should follow from it. Moreover, general poor test code quality and no tool support hinder the adoption of TDR.

Preprint: https://doi.org/10.5281/zenodo.2551217,

Data and materials: https://doi.org/10.5281/zenodo.4075356.

This chapter has been published *Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-Driven Code Review: An Empirical Study. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, may 2019.*[223]

8.1 Introduction

Peer code review is a well-established and widely adopted practice aimed at maintaining and promoting software quality [21]. In a code review, developers other than the code change author manually inspect a code change to find as many issues as possible and provide feedbacks that need to be addressed before accepting the code in production [168].

The academic research community is conducting empirical studies to better understand the code review process [21, 27, 29, 219, 224], as well as to obtain empirical evidence on aspects and practices that are related to more efficient and effective reviews [32, 33].

A code review practice that has only been touched upon in academic literature [20], but has been described in gray literature almost ten years ago [225] is that of *test-driven code review* (TDR, henceforth). By following TDR, a reviewer inspects a patch by examining the changed test code *before* the changed production code.

To motivate TDR, P. Zembrod—Senior Software Engineer in Test at Google—explained in the Google Blog [225]: "When I look at new code or a code change, I ask: What is this about? What is it supposed to do? Questions that tests often have a good answer for. They expose interfaces and state use cases". Among the comments, also S. Freeman—one of the ideators of Mocks [137] and TDD [6]—commented how he covered similar ground [226]. Recently, in a popular online forum for programmers, another article supported TDR (collecting more than 1,200 likes): "By looking at the requirements and checking them against the test cases, the developer can have a pretty good understanding of what the implementation should be like, what functionality it covers and if the developer omitted any use cases." Interviewed developers reported preferring to review test code first to better understanding the code change before looking for defects in production [20].

Despite these compelling arguments in favor of TDR, we have no systematic knowledge on this practice: its effectiveness in finding defects during code review, its prominence in practice, and what are its potential problems/advantages. This knowledge can provide insights for both practitioners and researchers. Developers and project stakeholders can use empirical evidence about TDR effects, problems, and advantages to make informed decisions about when to adopt it. Researchers can focus their attention on the novel aspects of TDR and challenges reviewers face to inform future research.

In this chapter, our goal is to obtain a deeper understanding of TDR. We do this by conducting an empirical study set up in two phases: An experiment, followed by an investigation of developers' practices and perceptions.

In the first phase, we study the effects of TDR in terms of the proportion of defects and maintainability issues found in a review. To this aim, we devise and analyze the results of an online experiment in which 92 developers (77 with at least two years of professional development experience) complete 154 reviews, using TDR or two alternative strategies (*i.e.*, production first or only production). Two external developers rated the quality of the review comments. In the second phase, we investigate problems, advantages, and frequency of adoption of TDR – valuable aspects that could not be studied in the experiment. To this aim, we conduct nine interviews with experiment participants and deploy an online survey with 103 respondents.

Key findings of our study include: With TDR, the proportion of functional defects (*bugs* henceforth) found in production code and maintainability issues (*issues* henceforth) found in test code does not change. However, TDR leads to the discovery of more bugs in

test code, at the expenses of fewer issues found in production code. The external raters judged the quality of the review comments as comparable across all review strategies. Furthermore, most developers seem to be reluctant to devote much attention to tests, as they deem production code more important; moreover applying TDR is problematic, due to widespread poor test quality (reducing TDR's applicability) and no tool support (not easing TDR).

8.2 Related Work

To some extent, TDR can be considered as an evolution of classical reading techniques [227], as it shares the general idea to guide code inspectors with software artifacts (*i.e.*, test cases) and help them with the code review task.

Scenario-based inspections. Among reading techniques, Porter & Votta [228] defined the *scenario-based* approach, based on scenarios that provide inspectors with more specific instructions than a typical checklist and focus on a wider variety of defects. They discovered that such technique is significantly more useful for requirements inspectors. Later on, Porter *et al.* [229, 230] and Miller *et al.* [231] replicated the original study confirming the results. Other studies by Fusaro *et al.* [232] and Sandahl *et al.* [233] reported contradictory results, however without providing explanations on the circumstances leading scenario-based code inspection to fail. A significant advance in this field was then provided by Basili *et al.* [234], who re-visited the original scenario-based as a technique that needs to be specialized for the specific issues to be analyzed. They also defined a new scenario-based technique called *perspective-based* reading: The basic idea is that different aspects of the source code should be inspected by inspectors having different skills [234]. All in all, the papers mentioned above, provided evidence of the usefulness of reading techniques; their similarities with TDR, give an interesting rationale on why TDR could bring benefits.

Ordering of code changes. Research on the ordering of code changes is also related to TDR. In particular, Baum *et al.* argued that an optimal ordering of code changes would help reviewers by reducing the cognitive load and improving the alignment with their cognitive processes [35], even though they made no explicit reference to ordering tests. This may give theoretical value to the TDR practice. Code ordering and its relation to understanding, yet without explicit reference to tests or reviews, has also been the subject of studies [235, 236].

Reviewing test code. Many articles on classical inspection (*e.g.*, [237, 238]) underline the importance of reviewing tests; however, they do not leave any specific recommendation. The benefits of reviewing tests are also highlighted in two case studies [239, 240]. Already in Fagan's seminal paper [5], the inspection of tests is discussed, in this case noting fewer benefits compared to the inspection of production code. Winkler *et al.* [241] experimented with writing tests during inspection and found neither large gains nor losses in efficiency and effectiveness. Elberzhager *et al.* [242, 243] proposed to use results from code reviews to focus testing efforts. To our knowledge, in academic literature TDR has been explicitly referred to only by Spadini *et al.* [20]. In a more general investigation on how test files are reviewed, the authors reported that some practitioners indeed prefer to review test code first as to get a better understanding of a code change before looking for defects in pro-

duction code. Our work builds upon the research on reviewing test code, by investigating how reviewing test code can(not) be beneficial for the whole reviewing process.

8.3 Methodology

In this section we describe the research questions and the methodology we follow to conduct our study.

8.3.1 Research Questions

This study has two parts corresponding to two research questions. In the first part, we design and run an experiment to investigate the effects of TDR on code review effectiveness. We measure the effectiveness as the ability to find bugs and maintainability issues during a code review (*i.e.*, the main reported goal of code review [21]). Hence, our first research question:

> **RQ$_1$.** Does the order of presenting test code to the reviewer influence the code review's effectiveness?

More formally, the goal of the experiment is to test the following null hypotheses:

H0*pc_mi*: For production code, there is no difference in the proportion of found maintainability issues between the different review practices (**TF** and **PF**).

H0*pc_bugs*: For production code, there is no difference in the proportion of found bugs between the different review practices (**TF** and **PF**).

H0*tc_mi*: For test code, there is no difference in the proportion of found maintainability issues between the different review practices (**TF** and **PF**).

H0*tc_bugs*: For test code, there is no difference in the proportion of the number of bugs found between the different review practices (**TF** and **PF**).

H0*review_time*: There is no difference in time required for a review between the review practices (**TF** and **PF**).

H0*review_quality*: There is no difference in the review qualities between **TF**, **PF**, and **OP**.

Subsequently, we investigate the prominence of TDR and developers' perception toward this practice, also focusing on problems and advantages. To do so, we conduct semi-structured interviews and deploy an online survey. Hence, our second research question:

> **RQ$_2$.** How do developers perceive the practice of Test-Driven Code Review?

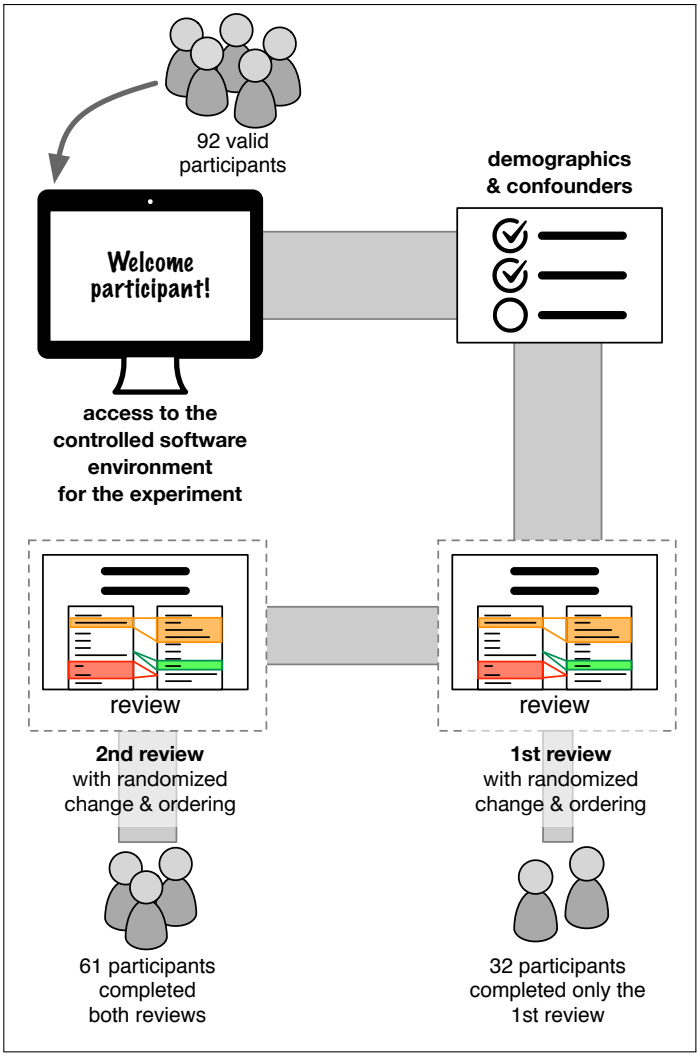


Figure 8.1: Experiment steps, flow, and participation

8.3.2 Method – RQ_1: Design Overview

Figure 8.1 depicts the flow of our experiment. We follow a partially counter-balanced repeated measures design [244], augmented with some additional phases.

1. We use a browser-based tool to conduct the experiment and answer $\mathbf{RQ_1}$. The tool allows to (i) visualize and perform code reviews, and (ii) collect data from demographic-like questions and the interactions that participants have with the tool. The welcome page provides information on the experiment to perform and requires informed consent.

2. After the welcome page, an interface is shown to collect demographics as well as information about some confounding factors such as: (i) the main role of the participant in software development, (ii) Java programming experience, (iii) current practice in programming and reviewing, and (iv) the hours already worked in the day of the experiment to approximate the current mental freshness. These questions are asked with the aim of measuring real, relevant, and recent experience of participants, as recommended by previous work [245]. Once filled in this information, the participant receives more details on the reviews to be performed.

3. Each participant is then asked to perform two reviews (the first is mandatory, the second is optional), randomly selected from the following three treatments[1] that correspond to the TDR practice and its two opposite strategies:

   - **TF** (*test-first*) – The participant must review the changes in both test code and production code, and is shown the changed test code first.
   - **PF** (*production-first*) – The participant must review both production and test, and is shown the production code first.
   - **OP** (*only-production*) – The participant is shown and must review only the changes to the production code.

   For the treatments **TF** and **PF**, the tool shows a 'Toggle Shown Code' button that allows the participant to see and review the other part of the change (*e.g.*, the production code change if the treatment is **TF**). We do not limit the number of times the 'Toggle Shown Code' button can be clicked, thus allowing the participant to go back and forth. The participant can annotate review remarks directly from the GUI of our tool.

4. Before submitting the experiment, we ask the participants if they would like to be further contacted for a semi-structured interview; if so, they fill in a text field with their email address. Further comments/impressions on the study can be reported using a text block.

8.3.3 Method – RQ_1: Browser-Based Experiment Platform

As previously mentioned, we adapt and use a browser-based experiment platform to run the experiment. This has two main advantages: On the one hand, participants can con-

[1] We propose two of the three treatments to keep the experiment as short as possible, thus stimulating a higher response rate, as also recommended by Flanigan *et al.* [246]. This choice does not influence our observations, as the random selection balances the treatments (see Table 8.4).

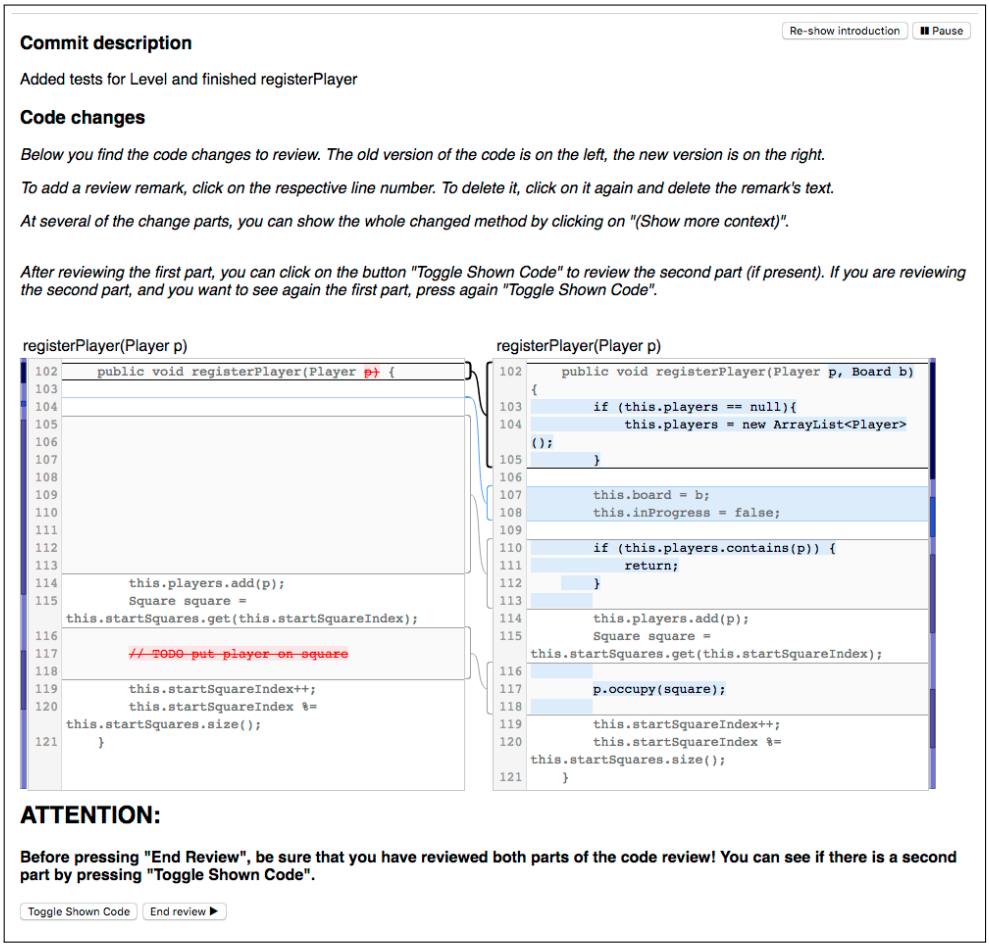


Figure 8.2: Example of the review view in the browser-based experiment UI, showing the code change. In this case, the treatment is **PF**, thus to see the test code, the participant must click on the 'Toggle Shown Code' button.

veniently perform code reviews; on the other hand, the tool assists us when gathering data from the demographic questions, conducting the different treatments, and collecting information for the analysis of co-factors. To reduce the risk of data loss and corruption, almost no data processing is done on the server: instead, participants' data is recorded as log records and analyzed offline.

The tool implements a GUI similar to other browser-based code review tools (*e.g.*, GitHub pull requests): It presents a code change in the form of two-pane diffs. An example of the implemented GUI is reported in Figure 8.2: Review remarks can be added and changed by clicking on the margin beneath the code. The tool logs many user interactions, such as mouse clicks and pressed keys, which we use to ensure that the participants are actively performing the tasks.

8.3.4 Method – RQ_1: Objects

The objects of the study are represented by the code changes (or *patch*, for brevity) to review, which need to be properly selected and eventually modified to have a sufficient number of defects to be discovered by participants.

Patches. To avoid giving some developers an advantage, we use a code base that is not known to all the participants of the experiment. This increases the difficulty of performing the reviews. To keep the task manageable, we ensure that (i) the functional domain and requirements are well-known to the participants and (ii) there is little reliance on special technologies or libraries. To satisfy these goals, we select an open-source project, **jpacman-framework**: The project consists of 42 production classes (≈2k LOC) as well as 13 test classes (≈600 LOC), it received 103 pull requests in his history, and has a total of 17 contributors.

To select suitable patches, we screen the commits of the project and manually select changes that (1) are self-contained, (2) involve both test and production code, (3) are neither too complicated nor too trivial, and (4) have a minimum quality, *e.g.*, not containing dubious changes.

The selected patches are those of commits *6d7c14d* and *698ac7d*. In the first one, a new feature is added along with the tests that cover it. In the second one, a refactoring in the production code is applied and a new test is added.

Seeding of bugs and maintainability issues. Code review is employed by many software development teams to reach different goals, but mainly (1) detecting bugs (functional defects) and (2) improving code quality (*e.g.*, finding maintainability issues), (3) spreading knowledge [21, 33, 180, 247].

Since the online experiment is done by single developers, we measure code review effectiveness by considering only the first two points, detecting bugs (functional defects) and maintainability issues. To this aim, we seed in the code bugs and maintainability issues. Examples of injected bugs are a wrong copy-paste and wrong boundary checking. For maintainability issues, we mainly mean *"smells that do not fail the tests"* [78], *e.g.*, a function that does more than what it was supposed to do, wrong documentation or variable naming. Concerning the nature of faults, one bug was real and identified some commits later. The other six were manually injected, based on standard errors such as handling corner cases and null pointer exceptions.

In the end, the two patches contain a total of 4 bugs and 2 issues (Patch 1) and 5 bugs and 3 issues (Patch 2). The total number of bugs per file (4 and 5) is higher than in the real-world code: indeed, in our experiment, we opted for a higher density of errors to ensure an attainable number of participants, and reduce confounding factors in the experiment. The reason is that if the number of errors in the code is too low, the statistical power decreases, and many more participants are needed to measure an effect. Moreover, if subjects must read too much correct code, the effects such as "reading speed," become stronger confounding factors.

8.3.5 Method – RQ_1: Variables and analysis

We investigate whether the proportion of defects found in a review is influenced by the review being done under a **TF**, **PF**, or **OP** treatment, controlling for other characteristics.

The first author of this chapter manually analyzed all the remarks added by the participants. Our tool explicitly asked and continuously highlighted to the participants that the primary goal is to find both bugs and maintainability issues; therefore, each participant's remark is classified as identifying either a bug or an issue, or as being outside of the study's scope. A remark is counted only if in the right position and correctly pinpointing the problem.[2]

By employing values at defect level, we could compute the dependent variables at review level, namely proportions given by the ratio between the number of defects found and the total number of defects in the code (dependent vars in Table 8.1). The dependent variables are then given by the average of n_j binary variables y_i, assuming a value 1 if the defect is found and 0 if not, where n_j is the total number of defects present in the change j, so that the proportion π_j results from n_j independent events of defect finding and y_j are binary variables that can be modeled through a logistic regression.

$$\pi_j = \sum_{1}^{n_j} \frac{y_j}{n_j} \tag{8.1}$$

The main independent variable of our experiment is the review strategy (or *treatment*). We consider the other variables as control variables, which include the time spent on the review, the review and programming practice, the participant's role, the reviewed patch (*i.e.*, P1 or P2), and whether the review is the first or the second being performed. In fact, previous research suggests the presence of a trade-off between speed and quality in code reviews [249]; following this line, we expect longer reviews to find more defects; to check this, we do not fix any time for review, allowing participants to perform the task as long as needed. Moreover, it is reasonable to assume that participants who perform reviews more frequently to also find a higher share of defects.

We run logistic regressions of proportions, where $Logit(\pi_j)$ represents the explained proportion of found defects in review j, β_0 represents the log odds of being a defect found

[2]To validate this coding process, a second author independently re-coded the remarks and compared his classification with the original one. In case of disagreements, the two authors opened a discussion and reached a consensus. We compute the Cohen's kappa coefficient [248] to measure the inter-rater agreement between the two authors before discussion: we find it to reach 0.9, considerably higher than the recommended threshold of 0.7 [248].

Table 8.1: Variables used in the statistical model

Metric	Description
Dependent Variables	
ProdBugsProp	Proportion of functional defects found in the *production* code
ProdMaintIssuesProp	Proportion of maintainability issues found in the *production* code
TestBugsProp	Proportion of functional defects found in the *test* code
TestMaintIssuesProp	Proportion of maintainability issues found in the *test* code
Independent Variable	
Treatment	Type of the treatment (TF, PF, or OP)
Control Variables	
Review Details	
TotalDuration	Time spent in reviewing the code
IsFirstReview	Boolean representing whether the review is the first or the second
Patch	Patch 1 or 2
Profile	
Role	Role(†) of the participant
ReviewPractice	How often(†) they perform code review
ProgramPractice	How often(†) they program
ProfDevExp	Years of experience(†) as professional developer
JavaExp	Years of experience(†) in Java
WorkedHours	Hours the participant worked before performing the experiment

(†) see Table 8.3 for the scale

for a review adopting **PF** (or **OP**) and of mean TotalDuration, IsFirstReview, *etc.*, while parameters $\beta_1 \cdot Treatment_j$, $\beta_2 \cdot TotalDuration_j$, $\beta_3 \cdot IsFirstReview_j$, $\beta_4 \cdot Patch_j$, *etc.* represent the differentials in the log odds of being a defect found for a change reviewed with **TF**, for a review with characteristics $TotalDuration_{j-mean}$, $IsFirstReview_{j-mean}$, $Patch_{j-mean}$, *etc.*.

$$\begin{aligned} Logit(\pi_j) = \beta_0 + \beta_1 \cdot Treatment_j + \beta_2 \cdot TotalDuration_j + \\ + \beta_3 \cdot IsFirstReview_j + \beta_4 \cdot Patch_j + \\ + ...(other\ vars\ and\ \beta\ omitted) \end{aligned} \tag{8.2}$$

8.3.6 Method – RQ_2: Data Collection And Analysis

While through the experiment we are able to collect data on the effectiveness of TDR, we cannot collect the perception of the developers on the prevalence of TDR as well as the

motivations for applying it or not. Hence, to answer $\mathbf{RQ}_2$, we proceed with two parallel analyses: We (i) perform semi-structured interviews with participants of the experiment who are available to further discuss on TDR and (ii) run an online survey with the aim of receiving opinions from the broader audience of developers external to the experiment.

Semi-structured interviews. We design an interview whose goal is to collect developers' points of view on TDR. They are conducted by the first author of this chapter and are semi-structured, a form of interview often used in exploratory investigations to understand phenomena and seek new insights on the problem of interest [183].

Each interview starts with general questions about code reviews, with the aim of understanding why the interviewee performs code reviews, whether they consider it an important practice, and how they perform them. Then, we ask participants what are the main steps they take when reviewing, starting from reading the commit message to the final decision of merging/rejecting a patch, focusing especially on the order of reviewing files. Up to this point, the interviewees are not aware of the main goal of the experiment they participated in and our study: We do not reveal them to mitigate biases in their responses. After these general questions, we reveal the goal of the experiment and we ask their personal opinions regarding TDR. The interview protocol is available [51].

During each interview, the researcher summarizes the answers and, before finalizing the meeting, these summaries are presented to the interviewee to validate our interpretation of their opinions. We conduct all interviews via SKYPE. With the participants' consent, the interviews are recorded and transcribed. Later, we analyze the interviews applying a Grounded Theory approach [250]: we use Descriptive Coding as first cycle coding, and Pattern coding as second cycle. We first summarize in a short phrase the essential topic of each passage from the interviews; then we identify explanatory codes to create emergent themes that we discussed among the authors. Overall, we conduct nine 20/30-minute interviews; Table 8.2 summarizes the demographics of the participants.

Table 8.2: Interviewees' experience (in years) and working context

ID	Developer	Reviewer	Working context	Applying TDR
P1	8	8	OSS	Almost never
P2	3	3	Company A	Almost Always
P3	15	15	Company A	Almost Always
P4	10	10	Company B	Almost Never
P5	10	5	Company C	Always
P6	3	2	Company D	Sometimes
P7	16	16	Company E	Always
P8	4	4	Company F / OSS	Sometimes
P9	3	3	Company G / OSS	Never

Online survey. We create an anonymous, 4-minute, online survey with two sections. In the first one, we ask demographic information of the participants, including gender, programming/reviewing experience, policies regarding code reviews in their team (*e.g.*, if all changes are subject of review or just a part of them), and whether they actually review test files (the respondents who answer "no" are disqualified). In the second section, we ask respondents (i) how often they start reviewing from tests vs. production files and

(ii) to fill out a text box explaining the reasons why they start from test/production files. The questionnaire is created using a professional tool [3] and is spread out through practitioners blogs (*e.g.*, Reddit) and through direct contacts in the professional network of the study authors, as well as the authors' social media accounts on Twitter and Facebook. Furthermore, we neither revealed the aim of the experiment nor provided incentives to participate.

We collected 103 valid answers, which complement the semi-structured interviews. Among the respondents, 5% have one year or less of development experience, 44% have 2 to 5 years, 28% have 6 to 10 years, and 23% more than 10 years.

8.4 Results – RQ_1: On The Effects Of TDR

Table 8.3: Participants' characteristics – descriptive statistics - n. 92

	Dev	Student	Researcher	Architect	Analyst	Other
Current role	61% (56)	16% (15)	12% (11)	5% (5)	3% (3)	2% (2)
Experience (years) with	**None**	**<= 1**	**2**	**3-5**	**6-10**	**>10**
- Java prog.	13% (12)	5% (5)	7% (6)	21% (19)	34% (31)	21% (19)
- Profess. dev.	5% (5)	11% (10)	13% (12)	18% (17)	28% (26)	24% (22)
Current frequency of	**Never**	**Yearly**	**Monthly**	**Weekly**	**Daily**	
- Programming	0% (0)	0% (0)	3% (3)	17% (16)	79% (73)	
- Reviewing	15% (14)	7% (6)	16% (15)	22% (20)	40% (37)	

A total of 232 people accessed our experiment environment following the provided link. From their reviews (if any), we exclude all the instances in which the code change is skipped or skimmed, by demanding either at least one entered remark or more than 5 minutes spent on the review. We also remove an outlier review that lasted more than 4 standard deviations from the mean review time, without entering any comments. After applying the exclusion criteria, a total of 92 participants stay for the subsequent analyses. Table 8.3 presents what the participants reported in terms of role, experience, and practice. Only 5 of the participants reported to have no experience in professional software development; most program daily (79%) and review code at least weekly (62%). Table 8.4 shows how the participants' reviews are distributed across the considered treatments and patches. Despite some participants completed only one review and the aforementioned exclusions, the automated assignment algorithm allowed us to obtain a rather balanced number of reviews per treatment and by patch.

Table 8.4: Distribution of participants' reviews across treatments

	TF	PF	OP	*total*
Patch1	31	32	29	*92*
Patch2	28	29	34	*91*
total	*59*	*60*	*63*	

[3] https://www.surveygizmo.com

8.4.1 Experiment results

Table 8.5 shows the average values achieved by the reviews for the dependent variables (*e.g.*, 'ProdBugsProp') and the average review time, by treatment. The most evident differences between the treatments are in: (d1) the proportion of maintainability issues found in production code (**PF** and **OP** have an average of .21 and .18, respectively, while **TF** of 0.08), and (d2) the proportion of bugs found in test code (**TF** has an average of 0.40, while **PF** of 0.17).

By applying a Wilcoxon Signed Rank Test [109] we tested and rejected H0pc_mi ($p <$ 0.01) and rejected H0tc_bugs ($p <$ 0.01). [4] On the contrary, based on the same test, we accept H0pc_bugs and H0tc_mi (the minimum p is higher than 0.38). [5] Applying again a Wilcoxon Signed Rank Test rejects H0$review_time$ ($p >$ .05 in all cases).

Overall, the comparison of the averages highlights that, within the same time, developers who started reviewing from tests (**TF**) spot a similar number of bugs in production, while discovering more defects in the tests but fewer maintainability issues in the production code. Thus, there seems to be a compromise between reviewing test or production code first when considering defects and maintainability issues.

Table 8.5: Average proportion of bugs and issues found, by treatment, and review time. Colored columns indicate a statistically significant difference between the treatments ($p <$ 0.01), with the color intensity indicating the direction.

	Proportion of found				Time
	production		test		
	bugs	maintIssues	bugs	maintIssues	
TF	0.28	**0.08**	**0.40**	0.18	7m11s
PF	0.33	**0.21**	**0.17**	0.13	6m27s
OP	0.28	**0.18**			5m29s

With regression modeling, we investigate whether these differences are confirmed when taking into account the characteristics of participants and reviews (variables in Table 8.1). In Section 8.6 we describe the steps we take to verify that the selected regression model is appropriate for the available data.

We build the four models corresponding to the four dependent variables, independently. Confirming the results shown in Table 8.5, the treatment is statistically significant exclusively for the models with 'ProdMaintIssuesProp' and 'TestBugsProp' as dependent variables; Table 8.6 reports the results.[6] We observe that—also considering the other factors—**TF** is confirmed a statistically significant variable in both 'ProdMaintIssuesProp' and 'TestBugsProp', with negative and positive directions, respectively. To calculate the odds of being a maintainability issue found in a review with **TF** compared to the baseline **OP**, we exponentiate the differential logit, thus: $exp(-1.25) = 0.29$, which means 71% fewer chances to find the issue in case of **TF** than **OP** (or **PF**). Instead, the odds of being a test bug found in a review with **TF** is 3.49, thus almost 250% more chances to find the test bug

[4] In the first case, Cliff's delta is medium, in the latter case, it is small

[5] It is arguable whether there is a need to apply a Bonferroni correction, because we tested the variable tc more than once. But even applying a correction necessarily leads to a p still < .05

[6] For space reasons, we omit the other two models, in which no variable is significant, but these models are available in our online appendix [51].

Table 8.6: Regressions for 'ProdMaintIssuesProp' and 'TestBugsProp'

	TestBugsProp			ProdMaintIssuesProp		
	Estimate	S.E.	Sig.	Estimate	S.E.	Sig.
Intercept	-0.7314	1.9242		-0.0471	1.2191	
TotalDuration	0.1664	0.0618	**	0.0462	0.0259	.
IsFirstReview 'TRUE'	-0.9213	0.5344	.	-0.1554	0.2241	
Patch 'P2'	1.9296	0.5688	***	-2.8474	0.4579	***
Treatment 'PF'				0.0975	0.2386	
Treatment 'TF'	1.1792	0.4639	**	-1.2468	0.3908	**
ReviewPractice	0.0675	0.2082		0.2598	0.1389	.
ProgramPractice	-0.6608	0.4685		-0.2180	0.2938	
ProfDevExp	-0.0982	0.1951		-0.2953	0.1211	*
JavaExp	-0.0182	0.1512		0.0366	0.0631	
WorkedHours	-0.0225	0.0817		0.0403	0.0405	
... (†)						

*significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1*
(†) Role is not significant and omitted for space reason

than with **PF**. Also, we see that the specific patch under review has a very strong significance for both models, thus confirming that differences in the code are an essential factor in the final outcome. The review time plays a significant role for 'TestBugsProp' and (to a lesser degree) for 'ProdMaintIssuesProp', in the expected direction. Unexpectedly, the professional experience plays a negative role for 'ProdMaintIssuesProp'; we hypothesize that professionals focus more on functional defects than maintainability issues.

In reviews in which tests are presented first, participants found significantly more bugs in test code, yet fewer maintainability issues in production code. The production bugs and test maintainability issues found is stable across the treatments.

8.4.2 Assessing code review quality

After having found differences in bugs/issues found with **TF**, we check whether the different treatments (**TF**, **PF**, **OP**) influence the quality of the review, i.e., we intend to test H0$review_quality$. To this aim two external validators manually classify each review, rating each comment. These validators have more than 5 years of industrial experience in code review and have collaborated in many open-source projects. We request them to go through each of the code reviews done by the developers involved in the experiment and rate each comment aimed at fixing a defect or maintainability issue in the production or test code using a Likert scale ranging between '1' (very poor comment) and '5' (very useful comment). They also give a score to each comment that is outside the scope (*i.e.*, a comment that does not fix any of our manually injected defects), plus a final overall score for the review. Each validator performs the task independently, and then their assessments are sent back to the authors of this chapter. The validators are unaware of the

treatment used in each review. A set of 33 reviews is classified by both authors, so that we can measure their inter-rater agreement using Cohen's kappa [248]. To mitigate the personal variability of the raters, we cluster their ratings into three categories: 'below average review comment', 'average review comment' and 'good review comment'. Then, we check the raters' agreement. The result shows that in all ratings there is at least a substantial agreement between the validators: production issues (κ = .69), test issues (κ = .78), production bugs (κ = 1), test bugs (κ = .77).

To test H0$review_quality$, we apply an ANOVA on the independent variables score (*e.g.*, productivity issues) and the dependent variable review practice (for each independent variable in separation). We find that for no score the independent variable is a significant factor (productivity issues p=.62; test issues p=.30; productivity bugs p=.25; test bugs p=.37). Hence, we accept H0$review_quality$ and conclude that raters do not see a difference in the quality of the reviews across treatments.

There is no statistically significant difference between the quality of the reviews made under the three considered treatments, according to two external raters blinded to the underlying treatments.

8.5 Results – $\mathbf{RQ}_2$: On The Perception Of TDR

We interview some of the participants of the experiment on what they perceive as advantages and disadvantages of TDR. We also survey 103 developers to enrich the data collection with people that do not participate in the experiment and can provide a complementary view on TDR. In this section, we report the answers obtained during our interviews and surveys. We summarize them in topics, covering both advantages and disadvantages of this practice. We refer to the interviewees by their ID shown in Table 8.2.

8.5.1 Adoption of TDR in practice

The majority of the respondents applies TDR only occasionally: 5% (5) reported that they always start from test code, 13% (13) almost always, 42% (43) occasionally/sometimes, 27% (28) almost never, 13% (13) never.

8.5.2 Perceived problems with TDR

When analyzing data coming from surveys and interviews, we discover a set of blocking points for the adoption of TDR. They can be grouped around four main themes, *i.e.*, perceived importance of tests, knowledge gained by starting reviewing from tests, test code quality, and code review tool ordering, that we further discuss in the following. Themes are discussed based on the frequency of appearance in the survey.

Tests are perceived as less important. From the comments left by the participants of the survey, it seems clear that test code is considered much less important than production code and that, as stated by one participant, they *"want to see the real change first"*. This

strong opinion is confirmed by other 15 participants;[7] for example, a participant explains that s/he starts *"looking at production files in order to understand what is being changed about the behavior of the system. [S/he] views tests as checks that the system is behaving correctly, so it does not seem possible to evaluate the quality of the tests until I have a clear understanding of the behavior"*. While the semi-structured interviews confirmed this general perception around tests, they also add a more practical point to the discussion: $P_{1,2-5,7}$ state that they need to prioritize tasks because of time, and often higher priority is given to the production code.

A closely related factor contributing to this aspect is the tiredness associated with reviewing code for a long time. As reported by one of the survey participants, *"the longer you are reviewing, the more sloppy you get [...]. I would rather have a carefully reviewed production file with sloppy test than vice versa"*. In other words, when performing multiple code review at once, developers often prefer to pay more attention to the production code than test files.

13 participants also report that is the production code to *drive* tests rather than the opposite. A clear example of this concept is enclosed in the following participant's quote: *"To me, tests are about checking that a piece of software is behaving correctly, so it doesn't make sense to me to try to understand if the tests are testing the right conditions if I do not understand what the code is supposed to do first."* Finally, another aspect influencing the perception that developers have of tests is the lack of testing experience. One of the participants affirms that *"not everyone in my team has lots of experience with testing, so usually just looking at the production code will tell me if there will be problems with the tests."* Thus, having poor experience in testing practice might bias the perception of the advantages given by tests in the context of code review.

Tests give less knowledge on the production behavior. Both interviewees and survey respondents report that the main advantage of starting with production code is that they can immediately see the feature and how it is changed, and only later they will check if it is properly tested. For example, a survey respondent says: *"I want to understand what the production code does, form my own opinion of what should be tested, and then look at the tests afterward."* From the interviews, P_9 also adds that it is hard to capture all the possible behaviors with tests, while looking to production code first helps him/her figuring out the failure modes before seeing what the tests the developer proposed are.

Nevertheless, an interesting trend emerges from our results. Despite most developers claim that tests cannot give enough knowledge to review a change, six of them declare that the decision of start reviewing from test code basically depends on the *degree* of knowledge they have of the production code under test. As explained by a survey participant, *"If I am familiar with the topic and the code, I will start with the production files. Otherwise I will choose the test files"*. In other words, tests only seem to be useful in time of need, *i.e.*, when a developer does not have any other instrument to figure out the context of the proposed code change.

Tests have low code quality. 4 participants mention poor test code quality as a reason to not apply TDR in practice. The use of tests in a code review has the prerequisite that such tests can properly exercise the behavior of the production code. One of the participants,

[7] It should be noted that in the survey we gave the possibility to leave open comments; having 15 or more participants agreeing on exactly the same theme indicates a noteworthy trend.

when explaining why s/he prefers starting from production code, reports that *"sometimes the tests are bad and it is easier to understand how the code behave by looking at the feature code"*. This is also confirmed by the semi-structured interviews: the main obstacle when reviewing tests first is the assumption that the test code is well written [$P_{1-3,5,6,9}$]. Both P_1 and P_9 said that most of the times they prefer to start from production because they assume developers do not write good tests. P_1 says *"Usually, I start from production and maybe the reason is that many of the projects where I worked do not have that many tests"*. Even if the tests are present, sometimes reviewers find them difficult to understand: According to P_6, *"... the test needs to be written in a clear way to actually understand what's going on."* The solution to this problem—according to our interviewees—is to impose test rules, *e.g.*, all the changes to the production code should be accompanied by tests [$P_{2,3,5}$]. P_5 says that *"if [tests] are not good, I will ask to modify them. If they are not even present, I will ask to add them and only after I will review the patch"*.

Code review tool ordering. The final disadvantage is related to a practical problem: current code review tools present code changes following an alphabetic order, meaning that most of the times developers review following such order. This is highlighted by 15 survey participants and confirmed by the interviewees. For example, P_3 says: *"If there is a front-end, we do not have integration tests for all the features, so sometimes I do manual testing. In this case, I would stick with the order of GitLab. I think GitLab present tests later than production, so I generally start from that"*. Interestingly, this point came up also from 35% of the survey respondents who do not apply TDR, that indicated they start from production code because they simply follow the order given by the code review tool. According to interviewees P_3 and P_4, they follow the order of GitHub because in this way they are sure to have reviewed all the files in the patch, while going back and forth from file to file may result in skipping some files.

Perceived problems with TDR: Developers report to (1) consider tests as less important than production, (2) not being able to extract enough knowledge from tests, (3) not being able to start a review from tests of poor quality, and (4) being comfortably used to read the patch as presented by their code review tool.

8.5.3 Perceived advantages of TDR

We identify two main themes representing the major perceived advantages of adopting TDR, *i.e.*, the concise, high-level overview tests give on the functionalities of production code and the ability of naturally improving test code quality. We discuss those aspects based on the frequency in the survey.

TDR provides a black-box view of production code. 18 of the respondents explain that the main advantage they envision from the application of TDR is the ability of tests to provide a concise, high-level overview of the functionalities implemented in the production code. In particular, one participant reports: *"If I read the test first without looking at the production implementation, I can be sure that the test describes the interface clearly enough that it can serve as documentation."* Moreover, developers appreciate that few lines of test code allow them to contextualize the change. Most interviewees agreed that starting re-

viewing tests allows them to understand better the code change context [P_{2-8}]. P_3 says: *"I think starting from tests helps you in understanding the context first, the design, the "what are we building here?" before actually looking at "how they implemented it".* The common feeling between the interviewees is that tests better explain what the code is supposed to do, while the production code shows how the developer implemented it [$P_{1-4,6,7}$]. P_1 says: *"When you are reviewing complex algorithms it is nice to immediately see what type of outputs it produces in response to specific types of input."*

TDR improves test code quality. Three of the survey participants explicitly report that tests must be of good quality and a practice like TDR would enable a continuous test code quality improvement; as one put it: *"Tests are often the best documentation for how the production code is expected to function. Getting tests right first also contributes to good TDD practices and the architectural values that come with that".* In other words, the developers report that, in situations where reviewers inspect tests first, the improvement of test code quality have to necessarily happen, otherwise reviewers could not properly use tests as documentation to spot problems in production code. As a natural consequence, TDR would also enforce tests to be updated, thus producing overall benefits to the system reliability.

Furthermore, one participant reports TDR to be efficient *"because it captures and should capture all the bugs"*: even if we obtain this kind of feedback by a small number of developers, it is still interesting to remark how some of them perceive the potential benefits of TDR, which we empirically found (**RQ**$_1$), as being more effective in terms of test bugs discovered—while spotting the same proportion of production bugs. The semi-structured interviews confirm all the aspects discussed so far. Most of the interviewees agreed that TDR somehow helps reviewers to be more focused on testing. According to P_9: *"I think it would encourage the development of good tests, and I think better tests mean more bugs captured. So yes, in the end, you may capture more bugs".* P_6 and P_8 also say that when reviewing the production code the reviewer already knows what the code is tested for, so it could be *easier* to catch not covered paths. For example, P_3 refers to happy *vs.* bad paths: *"starting from the tests you think a little bit on the cases that apply, so for example if they only test the happy path and not the bad path".*

Perceived advantages of TDR: Developers report that TDR (1) allows them to have a concise, high-level overview of the code under test and (2) helps them in being more testing-oriented, hence improving the overall test code quality.

8.6 Threats to validity

Construct validity. Threats to construct validity concern our research instruments. Most constructs we use are defined in previous publications, and we reuse existing instruments as much as possible: The tool employed for the online experiment is based on a similar tool used in an earlier work [51].

To avoid problems with the experimental materials, we employed a multi-stage process: After tests among the authors, we performed three pre-tests with external participants, and only afterward we started the release phase.

One of the central measures in our study is the number of defects and maintainability issues found. The first author seeded the defects, and later checked by the other authors. Nevertheless, we cannot rule out implicit bias in seeding the defects as well as in selecting the code changes.

We asked the participants to review test and production code separately, using a "Toggle shown code" button to switch between them. However, we cannot ensure that all the participants used this button correctly (or used it at all). To mitigate this, we analyzed the results mining only the shown code (*e.g.*, the second part of the review would not exist), obtaining very similar results: Hence we can conclude that this threat is not affecting the final results.

A major threat is that the artificial experiment created by us could differ from a real-world scenario. We mitigated this issue in multiple ways: (1) we used real changes, (2) we reminded the participant to review the files as they would typically do in their daily life, and (3) we used an interface very similar to the common Code Review Tools GUIs.

Furthermore, to validate the reviews done by the participants, we involved two external validators. The only information they had at their disposal when rating the reviews was the patch and the comments of the participants, *i.e.*, they did not know what treatment it was, the duration of the review, and all the other information we collected. Thus, the rate given by the validators was based on their personal judge and past experience in code review. To mitigate this issue, we involved two validators that have strong experience in Java and MCR: one had worked for many years in a large Russian-based SE company, and the other validator holds a Ph.D. in SE and has worked in many OSS.

Internal validity - Credibility. Threats to internal validity concern factors we did not consider that could affect the variables and the relations being investigated. In an online setting, a possible threat is the missing control over participants, which is amplified by their full anonymity. To mitigate this threat, we included questions to characterize our sample (*e.g.*, experience, role, screen size). To identify and exclude duplicate participation, we logged hashes of participant's local and remote IP addresses and set cookies in the browser. Furthermore, to exclude participants who did not take the experiment seriously, we excluded experiments without any comments in the review and manually classified the comments to delete the inappropriate ones.

We do not know the nature of the population that did our experiment, hence it might suffer from self-selection bias. Indeed, it could be possible that the sample contains better and more motivated reviewers than the population of all software developers. However, we do not believe this poses a significant risk to the validity of our main findings since we would expect stronger effects with a more representative sample. Furthermore, as depicted in Table 8.4, the participants' experience is quite various, with a 30% lower than 2 years and 50% with more than 6.

External validity - Transferability. Threats to external validity concern the generalization of results. A sample of 93 professional software developers is quite large in comparison to many experiments in software engineering [135]. However, it is still small compared to the overall population of software developers that employ MCR. We reduce this issue by interviewing and collecting the opinions of other developers 103 who did not participate in the experiment.

Statistical conclusion validity. A failure to reach statistically significant results is prob-

lematic because it can have multiple causes, *e.g.*, a non-existent or too small effect or a too small sample size. Even though we reached a quite large sample of participants, our sample is not large enough to detect smaller effects for RQ_1.

A major threat to our RQ_1 results is to employ the wrong statistical model. To ensure that the selected logistic regression model is appropriate for the available data, we first (1) compute the Variance Inflation Factors (VIF) as a standard test for multicollinearity, finding all the values to be below 1.5 (values should be below 10), thus indicating little or no multicollinearity among the independent variables, (2) run a multilevel regression model [251] to check whether there is a significant variance among reviewers, but we found little to none, thus indicating that a single level regression model is appropriate, (3) ascertain the linearity (assumed by logistic regression) of our independent continuous variable (the review time) and log odds using the Box-Tidwell test [252], and (4) build the models by adding the independent variables step-by-step and found that the coefficients remained stable, thus further indicating little to no interference among the variables.

Finally, in our statistical model, we control for the type of the patch, namely Patch 1 or 2. However, we do not control for Product or Process metrics of the code (*i.e.*, size, complexity, churn, etc.): we control only for the patch as it encloses all the characteristics that previous literature already demonstrated as related to review effectiveness [21, 180].

8.7 Discussion and Implications

Our findings provide two key observations to be further discussed and that lead to implications for practitioners, educators, tool vendors, and research community.

Ordering of files within the code review. Interviewees and survey respondents indicated that they often review the files as presented by their code review tool. While this process has the advantage that at the end a reviewer is sure to have reviewed all files, it may be problematic. For instance, our experiment (**RQ_1**) showed that simply presenting test first allows a reviewer to capture more test bugs, which have been shown to be extremely harmful to the overall reliability of software systems [69, 112]. At the same time, TDR still allows catching the same amount of bugs in production code, thus being nearly equivalent to the case of reviewing production files first. However, the drawback consists of finding fewer issues in production. A study could be designed and conducted to verify whether and to what extent using static analyzers could help to mitigate this drawback, as they can spot several maintainability issues automatically [199]. Furthermore, this finding can inform both next-generation review tools makers, which could base the order of files within the code review on the context, or allow the reviewers to make this choice.

We also found that developers decide on whether to start reviewing from test or production based on different factors such as familiarity with the code or type of modification applied. This suggests that to improve productivity and code review performance, tool vendors might enable the option to let developers decide on the code review ordering. At the same time, the research community is called to the definition of novel techniques that can exploit a set of metrics (*e.g.*, change type or past modifications of the developer on the code under review) to automatically recommend the order that would allow the reviewer to be more effective: this would lead to new adaptive mechanisms that take into account developer-related factors to improve the reviewability of code [30].

Test code quality. A critical enemy of TDR seems to be the poor quality of test code [12, 83]. Many interviewees and survey respondents indicated this as the main reasons to *not* apply TDR. If tests are poorly written or incomplete, it becomes almost impossible (or even dangerous) to start reviewing from test code, as it is harder to spot errors in production code. However, this would create a vicious cycle, in which tests are reviewed less and less carefully, thus gradually losing quality. Furthermore, we believe that the programming language can potentially influence TDR: in fact, the availability of testing frameworks (e.g., JUnit) affects this practice, and the readability of tests (which may also depend on the framework and language) can also do it. A possible solution consists of enforcing the introduction of code of conducts that explicitly indicate rules on how to review tests [253].

The development of a good team culture in which test code is considered as important as production code should be a must for educators. Indeed, as previous work already pointed out [20, 21, 180], good reviewing effectiveness is found mostly within teams that value the time spent on code review; hence, practitioners should set aside sufficient time for reviewing all the files present in the patch, including test code.

8.8 Conclusion

In this chapter, we empirically investigated a code review practice mentioned among practitioners' blogs and only touched upon in academia: Test-Driven Code Review. We performed a study set up in two phases: an experiment with 92 developers and two external raters, followed by a qualitative investigation with nine semi-structured interviews and a survey with 103 respondents. Among our results, we found that with TDR the quality of the review comments does not change and neither does the time spent and the proportion of found production bugs and test issues. TDR leads to the discovery of more bugs in test code, at the expenses of fewer maintainability issues found in production. We report that developers see the application of TDR as problematic because of the perceived low importance of reviewing tests, poor test quality, and no tool support. However, test first review is also deemed to offer a concise, high-level overview of a patch that is considered helpful for developers not familiar with the changed production code.

9

Primers or Reminders? The Effects of Existing Review Comments on Code Review

In contemporary code review, the comments put by reviewers on a specific code change are immediately visible to the other reviewers involved. Could this visibility prime new reviewers' attention (due to the human's proneness to availability bias), thus biasing the code review outcome? In this study, we investigate this topic by conducting a controlled experiment with 85 developers who perform a code review and a psychological experiment. With the psychological experiment, we find that ≈ 70% of participants are prone to availability bias. However, when it comes to the code review, our experiment results show that participants are primed only when the existing code review comment is about a type of bug that is not normally considered; when this comment is visible, participants are more likely to find another occurrence of this type of bug. Moreover, this priming effect does not influence reviewers' likelihood of detecting other types of bugs. Our findings suggest that the current code review practice is effective because existing review comments about bugs in code changes are not negative primers, rather positive reminders for bugs that would otherwise be overlooked during code review.

Data and materials: https://doi.org/10.5281/zenodo.3676517

This chapter has been published as *Davide Spadini, Gül Çalikli, and Alberto Bacchelli. Primers or reminders? The Effects of Existing Review Comments on Code Review. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering – ICSE 2020, pages 1171–1182, New York, NY, USA, jun 2020. ACM.* [254]

9.1 Introduction

Peer code review is a well-established practice that aims at maintaining and promoting source code quality, as well as sustaining development teams by means of improved knowledge transfer, awareness, and solutions to problems [14, 21, 29, 180].

In the code review type that is most common nowadays [34], the *author* of a code change sends the change for review to peer developers (also knowns as *reviewers*), before the change can be integrated in production. Previous research on three popular open-source software projects has found that three to five reviewers are involved in each review [20]. Using a software review tool, the reviewers and the author conduct an asynchronous online discussion to collectively judge whether the proposed code change is of sufficiently high quality and adheres to the guidelines of the project. In widespread code review tools, reviewers' comments are immediately visible as they are written by their authors; could this visibility bias the other reviewers' judgment?

If we consider the peer review setting for scientific articles, reviewers normally judge (at least initially) the merit of the submitted work independently from each other. The rationale behind such preference is to mitigate group members' influences on each other that might lead to errors in the individual judgments [255]. It is reasonable to think that also in code review, the visibility of existing review comments made by other developers may affect one's individual judgment, leading to an erroneous judgment.

An existing comment may prime new reviewers on a specific type of bug, due to the *availability bias* [256].

Availability bias is the tendency to be influenced by information that can be easily retrieved from memory (*i.e.*, easy to recall) [257]. This bias is one of the many cognitive biases identified in psychology, sociology, and management research [256]. Cognitive biases are systematic deviations from optimal reasoning [256, 258, 259]. In the cognitive psychology literature, Kahneman and Tversky showed that humans are prone to availability bias [260]. For example one may avoid traveling by plane after having seen recent plane accidents on the news, or may see conspiracies everywhere as a result of watching too many spy movies [257]. Therefore, it seems fitting to imagine that a reviewer may be biased toward a certain bug type, by readily seeing another reviewer's comment on such a bug type. This bias would likely result in a distorted code review outcome.

In this chapter, we present a controlled experiment we devised and conducted to test the current code review setup and reviewers' proneness to availability bias. More specifically, we examine whether priming a reviewer on a bug type (achieved by showing an existing review comment) biases the outcome of code review.

Our experiment was completed by 85 developers, 73% of which reported to have at least three years of professional development experience. We required each developer to conduct a code review in which an existing comment was either shown (treatment group) or not (control group). We then measured to what extent the reviewers could find—in the same code change—(1) another bug of the same type as the primed one and (2) a bug of a different type. We created a setup with two different code changes to review.

Based on the availability bias literature, we expected the primed participants (treatment group) to be more likely to find the bug of the same type (as it is already available in memory), but less likely to find the other bug type (since distracted by the comment).

9.2 Background and Related Work

In this section, we review the literature on scientific peer review. Subsequently, we provide background on cognitive biases in general and present relevant studies in Software Engineering (SE). We also provide a separate subsection on availability bias, which consists of some theoretical background and existing research on availability bias in SE.

9.2.1 Scientific peer review

Peer review is the main form of group decision making used to allocate scientific research grants and select manuscripts for publication. Many studies demonstrated that individual psychological processes are subject to social influences [261]. Such finding also points out some issues that might arise during group decision making. Experimental results obtained by Deutsch and Gerard [261] show that when a group situation is created, normative social influences grossly increase, leading to errors in individual judgment. Based on the findings of this study, it is emphasized that group consensus succeeds only if groups encourage their members to express their own, independent judgments. Therefore, one of the procedures for peer review of scientific research grant applications is 'written individual review' [255]. With this review procedure, reviewers judge the merit of a grant application in written form, independently of one another, before the final decision maker approves or rejects an application. Written individual review can mitigate the influence of reviewers on the way to reach a collective judgment. It is also used in scientific venues to eliminate biases. There is also another form of review procedure, namely panel peer review where a common judgment is reached through mutual social exchange [255]. In panel peer review, a group of reviewers convene to jointly deliberate and judge the merit of an application before the funding decision is made. However, as also emphasized by Deutsch and Gerard [261], it is crucial to encourage individual members to express their own judgment without feeling under the pressure of normative social influences for proper functioning of group decision making.

9.2.2 Cognitive biases in software engineering

Cognitive biases are defined as systematic deviations from optimal reasoning [256, 258, 259]. In the past six decades, hundreds of empirical studies have been conducted showing the existence of various cognitive biases in humans' thought processes [257, 258]. Although many theories explain why cognitive biases exist, Baron [262] stated that there is no evidence so far about the existence of a single reason or generative mechanism that can explain the existence of all cognitive bias types. Some theories see cognitive bias as the by-product of cognitive heuristics that humans developed due to their cognitive limitations (e.g., information processing power) and time pressure, whereas some relate them to emotions.

Human cognition is a crucial part of software engineering research since software is developed by people for people. In their systematic mapping study [256], Mohanini *et al.* report 37 different cognitive biases that have been investigated by software engineering studies so far. According to the results of this systematic mapping study, the cognitive biases that are most common in software engineering studies are *anchoring bias*, *confirmation bias*, and *overconfidence bias*. Anchoring bias results from forming initial estimates about a problem under uncertainty and focusing on these initial estimates without mak-

ing sufficient modifications in the light of more recently acquired information [257, 259]. Anchoring bias has so far been studied in software engineering research within the scope of requirements elicitation [263], pair programming [264], software reuse [265], software project management [266], and effort estimation [267]. Confirmation bias is the tendency to search for, interpret, favor, and recall information in a way that affirms one's prior beliefs or hypotheses [268]. The manifestations of confirmation bias during unit testing and how it affects software defect density have been widely studied in software engineering literature [269–271].

Any positive effect of experience on mitigation of confirmation bias has not been discovered so far [272]. However, in some studies, participants who have been trained in logical reasoning and hypothesis testing skills were manifested less tendency towards confirmatory behavior during software testing [272]. Ko and Myers identify confirmation bias among the cognitive biases that cause errors in programming systems [273]. Van Vliet and Tang indicate that during software architecture design, some organizations assign devil's advocate so that one's proposal is not followed without any questioning [274]. Overconfidence bias manifests when a person's subjective confidence in their judgement is reliably greater than the objective accuracy of such a judgement [275]. This bias type has been studied within the context of pair programming [264], requirements elicitation [276] and project cost estimation [277].

Availability bias. Availability bias is the tendency to be influenced by information that can be easily retrieved from memory (i.e., easy to recall) [257]. The definition of availability bias was first formulated by Tversky and Kahneman [260], who conducted a series of experiments to explore this judgmental bias. However, including these original experiments, many psychology experiments do not go beyond comparing two groups (i.e., controlled and test group) to differ in availability. To the best of our knowledge, in cognitive psychology literature, the only experiment providing evidence for the mediating process that manifests availability bias was devised by Gabrelcik and Fazio, who employed (memory) priming as the mediating process [278].

Availability bias has also been studied in SE research. De Graaf et al. [279] examined software professionals' strategies to search for documentation by using think-aloud protocols. Authors claim that using incorrect or incomplete set of keywords, or ignoring certain locations while looking for documents due to availability bias might lead to huge losses. Mohan and Jain [280] claim that while performing changes in design artifacts, developers—due to availability bias—might focus on their past experiences, since such info can be easily retrieved from developers' memory. However, such information might be inconsistent with the current state of the software system. Mohan et al. [280] propose traceability among design artifacts as a solution to mitigate the negative effects of the availability bias and other cognitive biases (i.e., anchoring and confirmation bias). Robins and Redmiles [281] propose a software architecture design environment reporting that it supports designers by addressing their cognitive challenges, including availability bias. Jørgensen and Sjøberg [282] argue that while learning from software development experience, learning from the right experiences might be hindered due to availability bias. Authors suggest retaining post-mortem project reviews to mitigate negative effects of availability bias.

Overall, existing literature points to the potential risks associated with availability bias in SE. As our community has provided evidence that code review is a collaborative and

cognitively demanding process and that the collaborative nature of code review also has the potential to affect individual reviewers' cognition, availability bias could manifest itself during the code review process. This bias could hamper code review effectiveness. In our study, we aim to explore how existing review comments bias the code review outcome.

9.3 Experimental Design

In this section, we explain the design of our experiment.

9.3.1 Research Questions and Hypotheses

The chapter is structured along two research questions. By answering these research questions, we aim to understand to what extent contemporary code review is robust to reviewers' availability bias, depending on the nature of the bug for which a previous comment exists on the code change. Our first research question and the corresponding hypotheses follow.

> **RQ$_1$.** *What is the effect of priming the reviewer with a bug type that is not normally considered?*

We hypothesize that an existing review comment about a bug type that reviewers do not usually consider (such as a `null` value passed as an argument [34, 283–285]) might prime the reviewers towards this bug type, so they find more of these bugs. Also, we hypothesize that—due to such priming—reviewers overlook bugs on which they were not primed. Hence, our formal hypotheses are:

$H0_{10}$**:** Priming subjects with bugs they usually do **not** consider does not affect their performance in finding bugs of the same type.

$H0_{11}$**:** Priming subjects with bugs they usually do **not** consider does not affect their performance in finding bugs they usually look for.

We also explore how priming on a bug that is usually considered during code reviews affects review performance. Therefore, our second research question is:

> **RQ$_2$.** *What is the effect of priming the reviewer with a bug type that is normally looked for?*

We hypothesize that also in the case of an existing review comment about a bug type that reviewers usually consider primes the reviewers towards this bug type, so that they find more of these bugs. Also, we expect primed reviewers to only look for the type of bugs on which they are primed, overlooking others. Hence, our formal hypotheses are:

$H0_{20}$**:** Priming subjects with bugs they usually consider does not affect their performance in finding bugs of the same type.

$H0_{21}$**:** Priming subjects with bugs they usually consider does not affect their performance in finding bugs they usually do **not** look for.

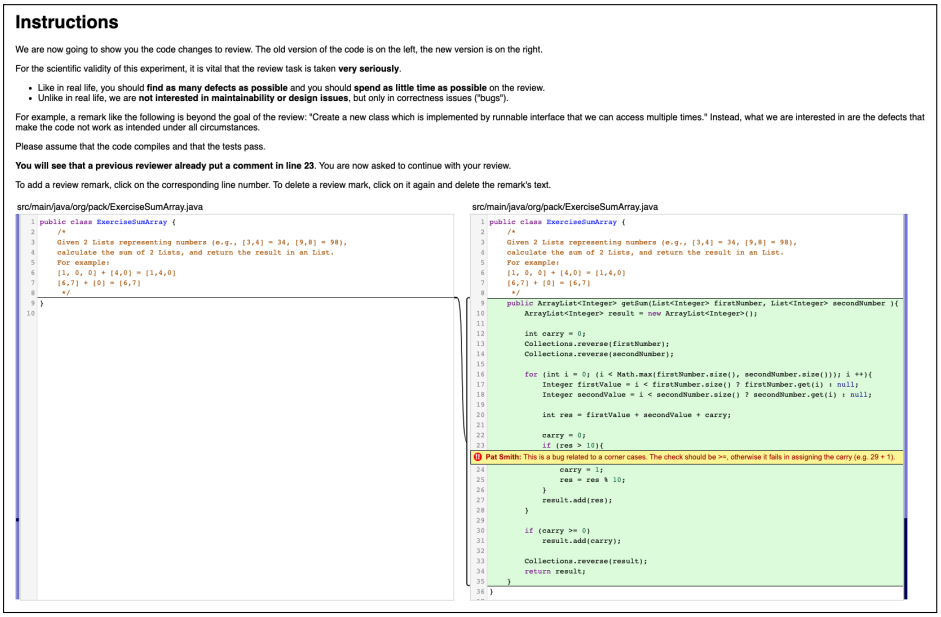


Figure 9.1: Example of a code review using the tool.

9.3.2 Experiment Design and Structure

To conduct the code review experiment and to assess participants' proneness to availability bias, we extend the browser-based tool CRExperiment [1]. The tool allows us to (i) visualize and perform a code review, (ii) collect data through questions asking for subjects' demographics information as well as data consisting of participants' interactions with the tool, (iii) collect data to measure subjects' proneness to availability bias, by using a memory priming set-up to trigger subjects' use of availability heuristic that is followed by a survey. Both the priming set-up and the survey are inherited from a classic experiment in cognitive psychology literature that was designed by Gabrielcik and Fazio [278].

Code Review Experiment Overview. For the code review experiment, we follow independent measures design [286] augmented with some additional phases. The following stages in the browser-based tool correspond to the code review experiment:

1. **Welcome Page:** The welcome page provides participants with information about the experiment. This page also aims to avoid *demand characteristics* [287], which are cues and hints that can make the participants aware of the goals of this research study leading to change in their behaviour during the experiment. For this purpose, we do not inform the participants about the full purpose of the experiment, rather they are only told that the experiment aims to compare code review performance under different circumstances. Before starting the experiment, the subjects are also asked for their informed consent.

[1] https://github.com/ishepard/CRExperiment

2. **Participants' Demographics:** On the next page, subjects are asked questions to collect demographic information as well as confounding factors, such as: (i) gender, (ii) age, (iii) proficiency in the English language, (iv) highest obtained education degree, (v) main role, (vi) years of experience in software development, (vii) current frequency in software development, (viii) years of experience in Java programming, (ix) years of experience in doing code reviews, (x) current frequency of doing code reviews, and (xi) the number of hours subjects worked that day. It is kept mandatory that subjects answer these questions before proceeding to the next page where they will receive more information about the code review experiment they are about to take part in. We ask these questions to measure subjects' real, relevant, and recent experience. Collecting such data helps us to identify which portion of the developer population is represented by subjects who take part in our experiment [245].

3. **Actual Experiment:** Each participant is then asked to perform a code review and is randomly assigned to one of the following two treatments:

   - **Pr** (*primed*)– The subject is given a code change to review where there exists a review comment (made by a previous reviewer) about a bug in the code. The test group of our experiment comprises the subjects who are assigned to this treatment.
   - **NPr** (*not–primed*)– The subject is given a code change to review. In the code change, there are no comments made by any other reviewers. The control group of our experiment comprises the subjects who are assigned to this treatment.

   More specifically, the patch to review contains three bugs: two of the same type (*i.e.*, BugA) and one of a different type (*i.e.*, BugB). In the **Pr** group, the review starts with a comment made by another reviewer showing that one instance of BugA is present. The participant is then asked to continue the review. In the **NPr** group, the review starts without comments. The comments shown to the participants in the **Pr** group were written by the authors, and the wording was refined with the feedback from the pilots (Section 9.3.5). Each participant is asked to take the task very seriously. More specifically, we ask them to find as many defects as possible and, like in real life, spend as little time as possible on the review. However, unlike in real life, we ask them not to pay attention to maintainability or design issues, but only in correctness issues ("bugs"). For example, we discard comments regarding variable namings or small refactorings.

4. **Interruptions during the Experiment:** Immediately after completing the code review, the participants are asked whether they were interrupted during the task and for how long.

5. **Follow-up Questions:** In the last page of the code review experiment, the participants are shown the code change they just reviewed together with the bugs disclosed: For each bug, we show it and explain why it is a defect and in what cases it might fail. Then, for each bug, we ask the participants to indicate whether they captured it in the review:

- If the participants found the bug and they belonged to the **Pr** group, we ask them to what extent the comment of the previous reviewer influenced the discovery of the bug (using a 5-point Likert scale).
- If the participants did not find the bug (independently whether they were in the **Pr** or **NPr** group), we ask them to elaborate on why they think they missed the bug.

Assessment of Proneness to Availability Bias. The code review experiment is followed by a set-up that primes participants' memory to trigger availability bias. This set-up serves as a mediating process to manipulate availability bias so that we can measure the extent to which each subject is prone to this type of cognitive bias. To measure this phenomenon, we inherited the test part of the controlled experiment of Gabrielcik and Fazio [278]. In the original experiment, the difference in the results of control and test groups showed that (memory) priming triggered the participants' availability biases. There are three reasons why we selected this experiment for assessing the proneness to availability bias: (i) To the best of our knowledge, it is the only experiment where the underlying cognition mechanism (*i.e.*, memory priming) that triggers availability bias is explicitly devised; (ii) memory priming mechanism is also employed in code review experiment to trigger participants' availability bias; and (iii) survey in the original experiment makes it possible to quantitatively assess participants' proneness to availability bias. Therefore, the remaining stages in the browser-based tool comprise the following:

1. **Welcome Page:** We provide a second welcome page in which, to avoid demand characteristics [287], the participants are told that they are about to participate in an experiment that aims to explore software engineers' attention by testing a set of visual stimuli, instead of the actual goal.

2. **Warm-up Session:** We proceed with a warm-up session in which participants are asked to focus on a series of 20 words flashing once each on the screen. The words are randomly selected from the English dictionary, and none of them contain the letter 'T'. Each word flashes for $300ms$. At the end of the warm-up, we ask the participants to write three words they have seen and recall, and to make a guess if they do not remember them.

3. **Actual Psychology Experiment:** After the warm-up, we proceed with the actual psychology experiment: this time, we show two series of 20 words, all of them including the letter 'T'. This time words flash at a faster rate, *i.e.*, $150ms$, to avoid that the participants consciously recognize that the words have the letter 'T' so often, which would bias their last task [278]. After each series, we ask the participant to write three words they have seen and recall, and to make a guess if they do not remember them.

4. **Measuring Proneness to Availability Bias:**The last task of the participants is to answer 15 questions, which ask to compare the frequency words for a given pair of letters in the English dictionary. For example, given the question "Do more words contain T or S", participants responded on a 9-point scale, with one end labeled

"Many more contain T" and the other "Many more contain S". Our main goal is to measure the extent to which each subject is prone to availability bias. Hence, in 5 of the 15 questions we ask whether in the English dictionary there are more words containing the letter 'T' or another random letter. As in the experiment of Gabrielcik and Fazio [278], we expect the participants to indicate that there are more words containing the letter 'T' (even though this is not the case) since they were primed in step 3. The other 10 questions are used to prevent the participants from understanding the actual aim of the study.

9.3.3 Objects

The objects of the study are represented by the code changes (or patch, for brevity) to review, and the bugs that we selected and injected, which must be discovered by the participants.

Patches. To avoid giving some developers an advantage, the two patches are not selected from open-source software projects, hence they are not known to any of the participants. To maintain the difficulty of the code review reasonable (after all, developers are used to review only the codebase on which they work every day), we screen many websites that offer Java exercises searching for exercises that are: (1) neither too trivial nor too complicated (based on our experience teaching programming to students), (2) self-contained, and (3) do not rely on special technologies or frameworks/libraries.

After several brain-storming sessions among the authors, only two exercises satisfied these goals and were selected.

Defects. Code review is a well-established and widely adopted practice aimed at maintaining and promoting software quality [21, 29]. There are different reasons on why developers adopt this practice, but one of the main ones is to detect defects [21]. Hence, in our experiment we manually seed bugs (functional defects) in the code. More specifically, we seed two different types of bugs: one that could cause a `NullPointerException` (BugA), and one that could cause the return of a wrong value (BugB).

The bugs were injected in the code as follows:

- In Patch$_1$, we inject two BugA and one BugB (the priming is done on BugA),
- In Patch$_2$, we inject two BugB and one BugA (the priming is done on BugB),

The `NullPointerException` (BugA) in the first change was on the passed parameters. As reported by white [34] and gray literature [283–285], developers are not used to check for this kind of errors in code review, because they expect the caller to make sure the parameters are not null: hence, we use it as the *not normally considered bug* that we investigate in RQ$_1$. Instead, BugA in the second change (RQ$_2$) does not regard a parameter, to make sure that it is bug type that normally developers look for in a review.

9.3.4 Variables and Measurement Details

We aim to investigate whether participants that are primed on a specific type of bug are more likely to capture only that type of bug. To understand whether the subjects did find the bug (*i.e.*, the value for our dependent variables), we proceed with the following steps:

1. the first author of this chapter manually analyzes all the remarks added by the participants (each remark is classified as identifying a bug or being outside of the study's scope), then

2. the authors cross-validate the results with the answer given by the participants (as explained in Section 9.3.2, after the experiment the participants had to indicate whether they captured the bugs).

In Table 9.1, we represent all the variables of our model. The main independent variable of our experiment is the treatment (**Pr** or **NPr**). We consider the other variables as control variables, which also include the time spent on the review, the participant's role, years of experience in Java and Code Review, and tiredness. Finally, we run a logistic regression model similar to the one used by McIntosh*et al.* [32] and Spadini *et al.* [223]. To ensure that the selected logistic regression model is appropriate for the available data, we first (1) compute the Variance Inflation Factors (VIF) as a standard test for multicollinearity, finding all the values to be below 3 (values should be below 10), thus indicating little or no multicollinearity among the independent variables, (2) run a multilevel regression model to check whether there is a significant variance among reviewers, but we found little to none, thus indicating that a single level regression model is appropriate, and, finally, (4) when building the model we added the independent variables step-by-step and found that the coefficients remained stable, thus further indicating little to no interference among the variables. For convenience, we include the script to our publicly available replication package [52].

Availability bias score. We calculate availability bias scores as in the original experiment by Gabrielcik and Fazio [278]. The frequency comparisons on the 9–point scale were scored by assignments of a value between +4 and −4. Positive numbers were assigned for ratings indicating that letter 'T' was contained in more words than the other letter, while negative numbers were assigned in favour of the other letter. We calculated the availability bias score for each participant as the average (and also median) of values for the 5 relevant questions.

9.3.5 Pilot Runs

As the first version of the experiment was ready, we started conducting pilot runs to (1) verify the absence of technical errors in the online platform, (2) check the ratio with which participants were able to find the injected bugs (regardless of their treatment group), (3) tune the experiment on the proneness to availability bias (in terms of flashing speed and number of words to ask), (4) verify the understandability of the instructions as well as the user interface, and (5) gather qualitative feedback from the participants. We conducted three different pilot runs, for a total of 20 developers. The participants were recruited through the professional network of the study authors to ensure that they would take the task seriously and provide feedback on their experience. No data gathered from the 20 participants to the pilot was considered in the final experiment.

After each pilot run, we inspected the results and the qualitative feedback we received and discussed extensively among the authors to verify whether parts of the experiment should have been changed. After the third run, the required changes were minimal, and we considered the experiment ready for its main run.

9.3.6 Recruiting Participants

The experiment was spread out through practitioners blogs and web forums (*e.g.*, Reddit) and through direct contacts from the professional network of the study authors, as well as the authors' social media accounts on Twitter and Facebook. We did not reveal the aim of the experiment. To provide a small incentive to participate, we introduced a donation-based incentive of five USD to a charity per valid respondent.

Table 9.1: Variables used in the statistical model.

Metric	Description
Dependent Variables	
FoundPrimed	The participant found the bug that was primed
FoundNotPrimed	The participant found the bug that was not primed
Independent Variable	
Treatment	Type of the treatment (**Pr** or **NPr**)
Control Variables	
Gender	Gender of the participant
EnglishLevel	English Level
Age	Age of the participant
LevelOfEducation	Highest achieved level of education
Role	Role of the participant
ProfDevExp	Years of experience as professional developer
JavaExp	Years of experience in Java
ProgramPractice	How often they program
ReviewPractice	How often they perform code review
ReviewExp	Years of experience in code review
WorkedHours	Hours the participant worked before performing the experiment
Tired	How tired was the participant at the moment of taking the experiment
Stressed	How stressed was the participant at the moment of taking the experiment
Interruptions	For how long the participant was interrupted during the experiment
TotalDuration	Total duration of the experiment
PsychoExpIsPrimed	Whether the participant was primed in the psychology experiment

(†) see Figure 9.2 for the scale

9.4 Threats to Validity

Construct Validity. Threats to construct validity concern our research instruments. To measure the extent to which subjects are prone to availability bias, we used the memory priming mechanism and the survey that was employed in an experiment designed and conducted by Gabrielcik and Fazio, in cognitive psychology literature [278]. Data obtained from the controlled experiment that Gabrielcik and Fazio conducted provide direct evidence that memory priming can be a mediating process to trigger availability bias. The remaining constructs we use are defined in previous publications, and we reuse the existing instruments as much as possible. For instance, the tool employed for the online experiment is based on similar tools used in earlier works [35, 223].

To avoid problems with experimental materials, we employed a multi-stage process: After tests among the authors, we conducted three experiments with ≈7 subjects each time (for a total of 20 pilots) with external participants. After each pilot session, we made corrections to the experiment based on the feedback from the subjects of the pilot, materials were checked by the authors one more time before we launched the actual experiment.

Regarding defects and code changes, the first author prepared the code changes and corresponding test codes as well as injecting the defects into these code changes. These were later checked by the other authors. Code change and corresponding test code were on the same page, and subjects had to scroll down to proceed to the next page of the online experiment. In this way, we aimed to ensure that subjects saw the test code. Test code were added to make the experiment closer to a real world scenario.

A major threat is that the artificial experiment created by us could differ from a real-world scenario. We mitigated this issue by (1) re-creating as close as possible a real code change (for example, submitting test code and documentation together with the production code), and (2) using an interface that is identical to the common Code Review tool Gerrit [2] (both our tool and Gerrit use Mergely [3] to show the diff, also using the same color scheme).

Internal Validity. Threats to internal validity concern factors that might affect the cause and effect relationship that is investigated through the experiment. Due to the online nature of the experiment, we cannot ensure that our subjects conducted the experiments with the same set-up (*e.g.*, noise level and web searches), however we argue that developers in real world settings also have a multi-fold of tools and environments. Moreover, to mitigate the possible threat posed by missing control over subjects, we included some questions to characterize our sample (*e.g.*, experience, role, and education).

To prevent duplicate participation, we adjusted the settings of the online experiment platform so that each subject can take the experiment only once. To exclude participants who did not take the experiment seriously, we screened each review and we did not consider experiments without any comments in the review, that took less than five minutes to be completed, or that were not completed at all.

Furthermore, several background factors (*e.g.*, age, gender, experience, education) may have impact on the results. Hence, we collected all such information and investigated how these factors affect the results by conducting statistical tests.

[2] https://www.gerritcodereview.com
[3] http://www.mergely.com/

External Validity. Threats to external validity concerns the generalizability of results. To have a diverse sample of subjects (representative of the overall population of software developers who employ contemporary code review), we invited developers from several countries, organizations, education levels, and background.

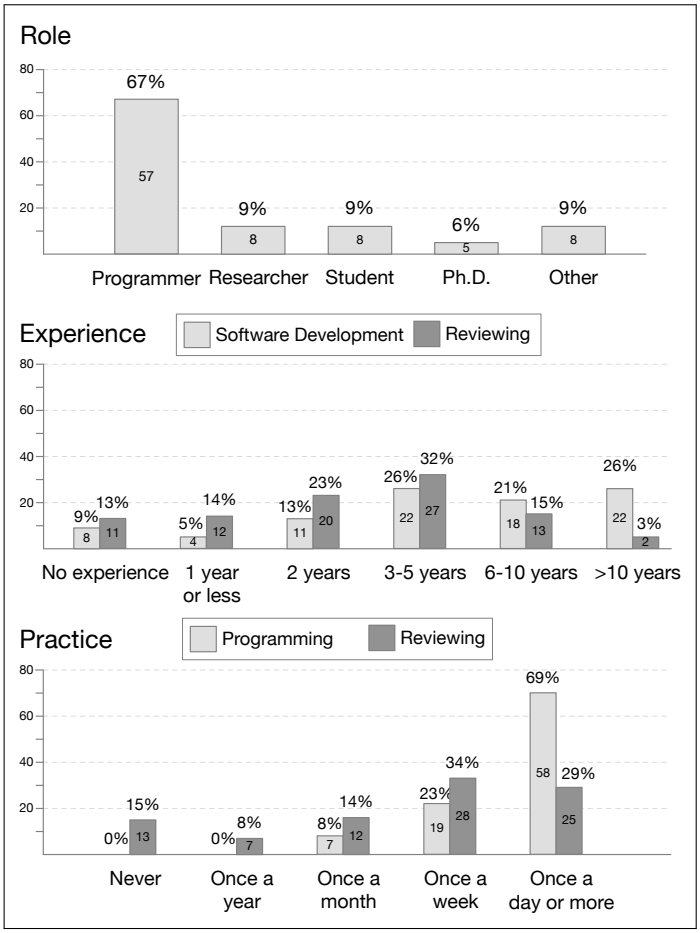


Figure 9.2: Participants' characteristics

9.5 Results

In this section, we report the results of our investigation on whether and how having a comment from a previous reviewer influences the outcome of code review.

9.5.1 Validating The Participants

A total of 243 people accessed our experiment environment following the provided link. From these participants, we exclude all the instances in which the code change is skipped

or skimmed, by demanding either at least one entered remark or more than five minutes spent on the review. After applying the exclusion criteria, a total of 85 participants are selected for the subsequent analyses.

Figure 9.2 presents the descriptive statistics on what the participants reported in terms of their role, experience, and practice. The majority of the participants are programmer (67%) and reported to have many years of experience in professional software development (73% more than 3 years, 47% more than 6); most program daily (69%) and review code at least weekly (63%).

Table 9.2 represents how the participants' are distributed across the considered treatments and code changes. The automated assignment algorithm allowed us to obtain a rather balanced number of reviews per treatment and code change.

Table 9.2: Distribution of participants (N = 85) across the various treatment groups.

	Primed (Pr)	**Not Primed (NPr)**	**Total**
CodeChange1	21	17	38
CodeChange2	22	25	47
Total	43	42	

9.5.2 RQ_1. Priming a not commonly reviewed bug

To investigate our first research question, the participants in our test group (**Pr**) are primed on a `NullPointerException` (NPE) bug in a method's parameter. We expect this type of bug to be missed by most not primed reviewer, because normally reviewers would assume that parameters are checked from the calling function [283–285].

Table 9.3 reports the results of the experiment by treatment group. From the first part of the table (primed bug), we can notice that participants in the **Pr** group found the other NPE bug 62% of the times, while participants in the **NPr** group only 11%. Expressed in odds, this result means that the NPE defect is 12 times more likely to be found by a participant in the **Pr** group. The main reasons reported by the participants in the **NPr** for missing this bug are that

1. they were too focused on the logic and not thoroughly enough when it comes the corner cases,
2. did not put attention to the fact that Integer could be null, and
3. that they generally do not check for NPE, but assume to not receive a wrong object as an input.

As expected, even though NullPointerException has been reported to be the most common bug in Java programs [288], developers stated they rarely sanity check the Object. However, as shown in Table 9.3, the result drastically changes when a previous reviewer points out that an NPE could be raised: in this case, many of the participants in the **Pr** group looked for other NPE bugs in the code.

When we look at whether the **Pr** group was primed by the previous reviewer comment (hence whether they were able to capture the bug because of they have been primed), we

Table 9.3: Odds ratio for capturing the primed and not primed bug in the test (Pr) and control (NPr) group.

Primed bug (NPE)	**Primed (Pr)**	**Not Primed (NPr)**	**Total**
found	13	2	15
not found	8	15	23
		Odds Ratio: 12.19 (2.19, 67.94)	
			$\boldsymbol{p < 0.001}$

Not primed bug	**Primed (Pr)**	**Not Primed (NPr)**	**Total**
found	14	14	28
not found	7	3	13
		Odds Ratio: 0.43 (0.09, 2.00)	
			$p = 0.275$

have that 40% indicated they were 'Extremely influenced', 40% were 'Very influenced' and 20% instead were 'Somewhat influenced'. Hence, the reviewers perceived to have been influenced by the existing comment.

We find a statistically significant relationship ($p < 0.001$, assessed using χ^2) of strong positive strength ($\phi = 0.5$) between the presence of the comment and whether the same type of bug was found. Therefore, we can reject $H0_{10}$.

Considering the second part of Table 9.3, we see that the not primed bug was found by both groups (**Pr** and **NPr**) at similar rate. For the former, participants found it 66% of the times, while in the **NPr** they found it 82% of the times. As shown in the table, the difference is not statistically significant ($p = 0.275$).

When looking at the participants' comments on why they missed this bug, we have that the main reasons are

1. that they forgot to try the specific corner case, and
2. that they assumed tests were covering all the corner cases.

The reasons for not capturing the defects were similar in both groups. Given this result, we cannot reject $H0_{11}$. Priming the participants on a specific type of bug did **not** prevent them from capturing the other type of bug.

In Table 9.4 we show the result of our statistical model, taking into account the characteristics of the participants and reviews. The model confirms the result shown in Table 9.3: even taking into account all the variables, the **isPrimed** variable is statistically significant exclusively for the primed bug. The other variable statistically significant in the model is 'Interruptions', that is the number of times the participant has been interrupted during the experiment: the estimate has a negative value, which means the higher the number of 'Interruptions', the lower the number of bugs captured, as one can expect.

For the not primed bug instead, none of the variables are statistically significant (with 'TotalDuration' and 'ReviewExp' are slightly significant, with $p < 0.1$)

Table 9.4: Regressions for primed and not primed bugs.

	Primed bug			Not primed bug		
	Estimate	S.E.	Sig.	Estimate	S.E.	Sig.
Intercept	0.704	4.734		-0.893	4.093	
IsPrimed	3.627	1.320	**	-1.199	1.073	
TotalDuration	0.001	0.002		0.003	0.001	.
ProfDevExp	0.813	0.557		-0.503	0.554	
ProgramPractice	-0.096	0.828		-0.243	0.736	
ReviewExp	-0.070	0.630		-0.813	0.651	
ReviewPractice	-1.152	0.758		1.243	0.643	.
Tired	-0.834	0.832		0.517	0.651	
WorkedHours	-0.069	0.196		0.305	0.207	
Interruptions	-1.752	0.758	*	-0.715	0.444	
... (†)						

*significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1*
(†) Role is not significant and omitted for space reason

Reviewers primed on a bug that is not commonly considered are more likely to find other occurences of this type of bugs. However, this does not prevent them in finding also other types of bugs.

9.5.3 RQ2. Priming on an algorithmic bug

To investigate our second research question, the participants in our test group (**Pr**) are primed on an algorithmic bug, more specifically a corner case (CC) bug. The result of this experiment is shown in Table 9.5. Participants in both groups found the primed bug ~50%. Indeed, the difference is not statistically significant ($p = 0.344$). If we consider whether the test group was primed by the previous reviewer comment, 50% of the participants reported that they were 'Extremely influenced', 10% was 'Somewhat influenced' and 40% was slightly or not influenced; thus suggesting that even the reviewers noticed a lower influence from this comment, even though it was of the same type as one of the other two bugs in the same code change.

Among the main reasons for missing the bug, participants mainly stated that

1. the tests drove them to not remember that corner case, and
2. they focused more on the first one.

Hence, given this result we can conclude that the participants who saw the review comment did **not** find the similar bug more often than the participants that did not see the review comment.

In the second part of Table 9.5, we indicate whether the participants were able to find the not primed bug. Both the test and control group are very similar in this case, too. Indeed, in both groups the bug is found around 50% of the times and the difference is not

statistically significant. When looking at the participants' comments on why they missed this bug, the main reasons they state are

1. that they were too focused on capturing algorithmic bugs without paying attention to NPE, and

2. that, as in the previous RQ, they did not put attention to the fact that `Integer` could be `null`.

Table 9.5: Odds ratio for capturing the primed and not primed bug in the test (Pr) and control (NPr) group.

Primed bug (CC)	**Primed (Pr)**	**Not Primed (NPr)**	**Total**
found	10	8	18
not found	12	17	29
		Odds Ratio: 1.77 (0.54, 5.81)	
		p = 0.344	

Not primed bug	**Primed (Pr)**	**Not Primed (NPr)**	**Total**
found	13	16	29
not found	9	9	18
		Odds Ratio: 0.81 (0.25, 2.64)	
		p = 0.73	

Given these results, we cannot reject $H0_{20}$ nor $H0_{21}$.

In Table 9.6, we show the result when controlling for other variables. Our dependent variable **IsPrimed** is not statistically significant. However, we see that 'TotalDuration' (*i.e.*, the time required by the developer to complete the code review) is statistically significant and in the expected direction. For the **NPr** group, the only variable that is significant is 'ReviewPractice' (*i.e.*, the average number of time the participant perform code reviews). Both these results are in line with what found in previous research [36].

Table 9.6: Regressions for primed and not primed bugs.

	Primed bug			**Not primed bug**		
	Estimate	S.E.	Sig.	Estimate	S.E.	Sig.
Intercept	-1.0510159	2.2460623		-3.037e-01	2.568e+00	
IsPrimed	0.9260383	0.7223408		-1.670e-01	7.740e-01	
TotalDuration	0.0018592	0.0008958	*	9.561e-05	9.976e-04	
ProfDevExp	-0.6031309	0.3381302	.	-9.437e-02	3.721e-01	
ProgramPractice	0.0319636	0.5905427		-1.061e+00	7.353e-01	
ReviewExp	0.3411589	0.4548836		1.284e-01	4.660e-01	
ReviewPractice	0.1531502	0.3784472		1.211e+00	4.683e-01	**
Tired	0.0835410	0.3706085		2.486e-01	4.539e-01	
WorkedHours	-0.1619234	0.1184626		2.257e-01	1.542e-01	
Interruptions	-0.1755182	0.3220796		-1.331e-01	3.630e-01	
... (†)						

*significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1*
(†) Role is not significant and omitted for space reason

Reviewers primed on an algorithmic bug perceive an influence, but are as likely as the others to find algorithmic bugs. Furthermore, primed participants did not capture fewer bugs of the other type.

9.5.4 Robustness Testing

In the previous sections, we presented the results of our study on whether and to what extent reviewers can be primed during code review by showing an existing code review comment. Surprisingly, the results showed that many of our hypotheses were not satisfied: in our experiment, only in one case primed reviewers captured more bugs than the not primed group; in all the other cases, reviewers from both groups could capture the same bugs.

To further challenge the validity of these findings, in this section, we employ *robustness testing* [289]. For this purpose, we test whether the results we obtained by our baseline model hold when we systematically replace the baseline model specification with the following plausible alternatives.

Bugs were too simple or too complicated to find. Choosing the right defects to inject in the code change is fundamental to the validity of our results. If a defect is too easy to find, participants might find the bugs regardless of any other influencing factor, even without paying too much attention to the review (on the other hand, if it is too complicated reviewers might not find any bug and get discouraged to continue). We measure that ~50% of the participants found the three types of defects that we expected them to find, thus ruling out the possibility that these bugs were either too trivial or too difficult to find.

People were not primed. The entire experiment is based on the premise that reviewers in the **Pr** group were correctly primed. Even though we can not verify this premise (the experiment is online, hence there is no interaction between the researchers and the participants), after the code review experiment the participants had to indicate whether they were influenced by the comment of the previous reviewer in capturing the bug. As we stated in Section 9.5.2 and Section 9.5.3, 70% of the participants indicated they were extremely or very influenced, while only 18% indicated somewhat or slightly influenced (12% were neutral). This gives an indication that the participants felt they were indeed primed, but this did not influence their ability to find other bugs.

Nevertheless, the reported level of being influenced is subjective, so not fully reliable (participants could think to have been influenced, but were not). To triangulate this result, we test another possibility: More specifically, one of the possible explanations of why participants may not have been primed is that our sample of participants was "immune" to priming or very difficult to prime. Indeed, there is no study that confirms that developers are as affected by priming as the general population (on which past experiment was conducted). To rule out this possibility, we devised the psychological experiment: We tested whether developers can also be primed as expected using visual stimuli. Our results show that ~70% of the participants were primed as expected.

Not enough participants. Another possibility of why we do not find a difference is that we did not have enough participants. Even though 85 participants is quite large in comparison to many experiments in software engineering [36] and we tried to design an

experiment that would create a strong signal, we cannot rule out that the significance was missing due to the number of participants. However, even if the results were statistically significant (assuming we had the same ratios, but an order of magnitude more of participants), the size of the effect (calculated using the ϕ coefficient) would be 'none to very negligible'. This suggests that there was no emerging trend and that, even having more participants, we could have probably obtained a significant, yet trivial effect.

Some participants did not perform the task seriously. Finally, one of the reasons why we did not confirm most of our hypotheses could be that some participants did not take the task seriously, hence they might have performed poorly and have altered the results. Having used a random assignment and having a reasonably large number of participants, we have no reason to think that one group had more 'lazy' participants than the others. Moreover, as we discussed in Section 9.3, to exclude participants who did not take the experiment seriously, we filtered out experiments without any comments in the review (even if there were comments, the first author manually validated them to check whether they were appropriate and they were/not capturing a bug); we also did not consider reviews that took less than five minutes to be completed, or that were not completed at all (maybe because the participant left after few minutes).

Alternatively, it would be possible that participants who were more serious focused more and found more bugs (regardless of the priming), while less serious ones would just find one and leave the experiment. To test also this possibility, we compared the likelihood of a participant in finding a second bug when a first one was found. Also in this case, we did not find any statistically significant effect, thus ruling out this hypothesis as well.

9.6 Discussions

We discuss the main implications and results of our study.

Robustness of code review against availability bias. The current code review practice expects reviewers to review and comment on the code change asynchronously, and reviewers' comments are immediately visible to both the author and other reviewers.

One of the main hypotheses we stated in our study is that the code review outcome is biased because reviewers are primed by the visibility of existing comment on a bug. Indeed, if reviewers get primed by previously made comments about some bug(s), then they could find more bugs of that specific type while overlooking other types of bugs. This would, in turn, undermine the effectiveness of the code review process, creating a demand for a different approach.

To create a different approach, one might consider adopting a review method similar to that of scientific venues where reviewers do not see the comments of the other reviewers until they submit their review. Even though this strategy would reduce the transparency of the code review process undermining knowledge transfer, team awareness, as well as shared code ownership, and would probably lead to a loss in review efficiency due to duplicate bug detection, it would be necessary if the biasing effect of other reviewers' comments would be strong.

Our experiment results show that the participants in the test group were *positively* influenced by the existing comment on the code change so that they could capture more bugs of the same type. However, unexpectedly, they were still able to capture the bugs of

the different type as the control group did. Like any human, reviewers are also prone to availability bias [257] to various extents. However, our results did not find evidence of a strong negative effect of reviewers' availability bias. Therefore, our data does not provide any evidence that would justify a change in the current code review practices.

Existing comments on normally not considered bugs act as (positive) *reminders* rather than (negative) *primers*. Surprisingly, participants in the test group who were primed with the algorithmic bug type (more specifically, a corner case bug) detected the same amount of corner case and `NullPointerException` (NPE) bugs as the participants in the control group. However, participants who were primed with a bug that is normally not considered in review (*i.e.*, NPE) were 12 times more likely to capture this type of bug, than the participants of the control group.

This result shows that existing reviewer comments on code change seem to support recalling (*i.e.*, act as a reminder), rather than distracting the reviewer. As previously mentioned in section 9.5.2, participants in the test group indicated that they were focused on to the corner cases in the code change and did not put attention to the possibility that *Integer* could be *null*. Such feedbacks are in-line with the possible existence of *anchoring bias* [257, 259].

It is likely that the existence of a reviewer comment on a uncommon bug had a debiasing effect on the participants in the test group (*i.e.*, mitigated the participants' bias). In software engineering literature, there are empirical studies on practitioners' anchoring bias. For instance, Pitts and Brown [263] provide procedural prompts during requirements elicitation to aid analysts not anchoring on currently available information. According to the findings by Jain *et al.* [264], pair programming novices tend to anchor to their initial solutions due to their inability to identify such a wider range of solutions.

However, to the best of our knowledge, there are no studies on anchoring bias within the context of code reviews. Therefore, further research is required to investigate underlying cognition mechanisms that can explain why existing reviewer comments on the unexpected bug act as *reminders*.

9.7 Conclusions

In the study presented in this chapter, we investigated robustness of peer code review against reviewers' proneness to availability bias. For this purpose, we designed and conducted an online experiment with 85 participants, including a code review task and a psychological experiment. With the psychological experiment, the majority of the participants (i.e., ≈70%) were assessed to be prone to availability bias (median = 3.8, max = 4). However, when it comes to the code review, our experiment results show that participants are primed only when the existing code review comment is about a type of bug that is not normally considered; when this comment is visible, participants are more likely to find another occurrence of this type of bug. Hence, existing comments on this type of bugs acted as *reminders* rather than *primers*. It is our hope that this study is replicated by other researchers to gain further insights about the extent of robustness of peer code review.

10

Conclusion

The goal of this dissertation is to discover new techniques and tools to support developers when writing and reviewing code. To this aim, we investigate the impact of test design issues on code quality, as well as practices when writing and reviewing test code.

10.1 Contributions

In this dissertation, we make the following main contributions:

In Chapter 2, we contribute to the scientific community with:

- we present Pydriller, an open-source Python framework for MSR. Pydriller is used by many researchers as well as companies such as SIG and Mozilla. Currently, Pydriller is downloaded ≈80K per month, it has 13 contributors and ≈380 stars on GitHub.

In Chapter 3 we present an investigation on the relation between test design issues and software code quality. This results in:

- evidence that test smells are associated with higher change- and defect-proneness of the affected test cases.
- evidence that 'Indirect Testing', 'Eager Test', and 'Assertion Roulette' smells are are associated with higher change- and defect-proneness.
- novel evidence that the presence of test smells contributes to the explanation of the defect-proneness of production code.
- evidence that 'Indirect Testing' and 'Eager Test' smells are associated with higher defect-proneness in the exercised production code.

In Chapter 4 we present an investigation on severity thresholds for test smells, to understand what are the ones with the highest refactoring priority. This results in:

- a new set of calibrated thresholds such that severity levels can be assigned to test smell instances
- evidence on which test smell instances have the highest impact on maintainability and the highest priority in refactoring according to the participants of our study
- evidence on why developers prefer not to refactor certain test smell instances

- recommendations on how to improve the test smells detection and better present the positive instances to developers

In Chapter 5 we present an investigation on how developers use Mock objects when writing tests, their current practices, and the relation between mocking and code quality. This results in:

- a categorization of the most often (not) mocked dependencies, based on a quantitative analysis on three OSS systems and one industrial system.
- an empirically-based understanding of why and when developers mock, based on interviews with developers of the analyzed systems and an online survey.
- a list of the main challenges when making use of mock objects in the test suites, also extracted from the interviews and surveys.
- an understanding of how mock objects evolve and, more specifically, empirical data on when mocks are introduced in a test class, and which mocking APIs are more prone to change and why.
- evidence on the relation between the usage of mocks and the code quality of the mocked class.
- an open source tool, namely MockExtractor, that is able to extract the set of (non) mocked dependencies in a given Java test suite.

In Chapter 6 we present an investigation on the developers' needs when doing code review. This results in:

- the identification of 7 high-level categories of information needs and their occurrences across 900 code reviews pertaining to three large open-source systems
- an investigation of how developers perceive the previously identified needs, by means of interviews with the developers of the analyzed systems
- an analysis of what needs are the most discussed, how much time it requires the reviewers to conclude the discussion and whether reviewers needs change during the lifespan of the code review

In Chapter 7 we present an investigation on how and why developers perform code review on test code. This results in:

- evidence that test files are as likely as production files to contain defects, suggesting they should benefit from code review.
- evidence that test files are not discussed as much as production files during code reviews, especially when test code and production code are bundled together in the same review.
- an analysis of the challenges faced by developers when reviewing test files, including dealing with a lack of testing context, poor navigation support within the review, unrealistic time constraints imposed by management, and poor knowledge of good reviewing and testing practices by novice developers.
- GerritMiner, an open source tool that extracts code reviews from projects that use Gerrit; we designed this tool to help us collect a dataset of 654,570 code reviews from three popular open source, industry-supported software systems.

In Chapter 8 we present an empirical study on TDR, based on the results of a controlled experiment and qualitative analysis. This results in:

- evidence that TDR leads to the discovery of more bugs in test code
- evidence that with TDR the proportion of bugs found in production code does not change.
- evidence on the perceived disadvantages of TDR, such as lack of support for this practice, low test code quality, and production code having the highest priority during code review.
- evidence on the perceived advantages of TDR, such as allowing developers to have a high-level overview of the code under test and helping them in improving the overall test code quality.

In Chapter 9 we present an in-depth analysis, based on a controlled experiment, of the impact of availability bias in code review. This results in:

- CREXPERIMENT, a browser-based tool that allows us to (i) visualize and perform a code review, (ii) collect data through questions asking for subjects' demographics information, and (iii) collect data to measure subjects' proneness to availability bias
- evidence that, for three out of four bugs, the outcome of the review is biased in the presence of an existing review comment.
- evidence that the reviewers were more likely to find a type of bug that is normally not considered during code review when they were primed on the same type of bug, indicating that existing review comments do not act as negative *primers*, rather as positive *reminders*

10.2 Reflection on the Research Questions

In light of our findings, in this section we try to answer the research questions defined in Chapter 1.

> **Research Question 1.** What is the impact of poor test design on code quality and how can we present this feedback to developers?

Our analysis to answer this research question exposes that a relation between test code design issues and software quality exists and that developers, in some cases, still do not perceive this situation as an actual problem. We determine this by first analyzing test smells instances of ten OSS projects and their software code quality. Then, we modify a test smells detector and we integrate it into a prototype extension of BCH [44], asking existing users of BCH to give feedback on their test code quality.

We observe that a test with design issues has 81% higher risk of being defective than a test without issues, and 47% higher risk of being changed. Moreover, we found that test methods with co-occurring smells tend to be more change-prone than methods having fewer smells and that 'Indirect Testing', 'Eager Test', and 'Assertion Roulette' are those associated with the most change-prone test code. When studying the impact of test smells on production code quality, we found that the presence of design flaws in test code is associated with the defect-proneness of the exercised production code; indeed the production code is 59% more likely to contain defects when tested by smelly tests, and that 'Indirect Testing' and 'Eager Tests' are related to a higher defect-proneness in production code.

Having found a relationship between the presence of test design issues and software code quality, we modified an existing open-source test smell detector and integrated it into BCH. First, we modified the detector with new thresholds to lower the amount of false positives: we performed a study on 1,489 projects to define new thresholds for the detection of nine test smells. Then, we integrated the detector in BCH and received feedback from 31 BCH users. The developers reported that 'Empty Test' and 'Sleepy Test' are the smells with the highest priority in refactoring, followed by 'Mystery Guest' and 'Resource Optimism'. The smells with the lowest priorities instead were 'General Fixture', 'Ignored Test', and 'Eager Test'. When analyzing the cases in which the developers rated an instance as "not important to fix", we saw that in the majority of cases they dismissed the test smell instance because they were unwilling to refactor the test (even if it was an easy fix). Even though by fixing the smell the readability and maintainability would be significantly improved, they perceived them to be marginal gains, thus concentrates their efforts elsewhere.

By addressing this research question we found that test design issues do have an impact on the overall software code quality of the system, thus we argue that developers should be careful when refactoring test code. Unfortunately, developers do not always perceive these issues as important and often are unwilling to spend time in refactoring their test code to fix the test smells. Additionally, better tools needs to be developed to fully support the detection of test smells: we took a first step into this direction with the creation of new thresholds, but more research is needed to lower the amount of false positive instances and increase the adoption of refactoring.

Implications. We showed that test design issues have an impact on the overall software system quality. From previous research we also know that bugs in test files can lower the quality of the corresponding production code, because a bug in the test can lead to serious issues such as 'silent horrors' or 'false alarms' [112]. Unfortunately, even with all these results, we noticed that developers in some cases still do not perceive test issues as an actual problem and choose to not refactor it.

As such, it represents a call to arms to researchers and tool vendors. We call upon researchers to further investigate the interplay between test design quality and the effectiveness of test code in detecting defects. We need to better understand what is the role of test code on the overall quality of the system, and how to present these results to developers. Furthermore, even if the research community brings knowledge on the importance of writing good test code, developers need tools to analyze the quality of their systems. Hence, we call upon tool vendors to develop practical automatic test issues detection tools. The tools that we used in our studies are still in the early stages of development: they are not practical (*e.g.*, the majority only works for Java systems), relatively slow, and often return false positives. For these reasons, developers do not trust these tools and, as a consequence, they do not use them. Tool makers need to improve the reliability of the test issues detection, so that developers can start to use them and refactor their test code.

Research Question 2. How do developers use Mocks in test code and why? Do mocks help in achieving an higher software quality?

In this RQ we sought to understand how developers use mocks in their software systems. To this aim, we performed a two-phase study. In the first part, we analyzed more than 2,000 test dependencies from three OSS projects and one industrial system, we interviewed developers from these systems to understand why they mock some dependencies and they do not mock others, and we challenged and supported our findings by surveying 105 developers from software testing communities. Finally, we discussed our results with a leading developer from the most used Java mocking framework.

In this first part, we found that classes related to external resources (*e.g.*, databases and web services) are often mocked, due to their inherently complex setup and slowness. Domain objects, on the other hand, did not display a clear trend concerning mocking, and developers tend to mock them only when they are too complex. Among the challenges, a major problem is maintaining the behavior of the mock compatible with the original class (*i.e.*, breaking changes in the production class impact the mocks). Furthermore, participants stated that excessive use of mocks may hide important design problems and that mocking in legacy systems can be complicated.

In the second part of the study, we analyzed the evolution of the mock objects as well as the coupling they introduce between production and test code. Our study showed that mocks are almost always introduced when the test class is created (meaning that developers opt for mocking the dependency in the very first test of the class) and mocks tend not to be removed from the test class after they are introduced. Furthermore, our results showed that mocks change frequently. The most important reasons that force a mock to change are (breaking) changes in the production class API or (breaking) changes in the internal implementation of the class, followed by changes solely related to test code improvements (refactoring or improvements).

With this research question, we brought empirical evidence on what practices developers follow when using mocks in their tests. We argue that these practices can help developers to write tests of better quality, and inform tool makers about which features practitioners really need (and do not need) in practice.

Implications. By providing a deeper investigation on how and why developers use mock objects, as a side effect, we also notice how the use of mock objects can drive the developer's testing strategy. For instance, mocking an interface rather than using one concrete implementation makes the test to become "independent of a specific implementation", as the test exercises the abstract behavior that is offered by the interface. Without the usage of a mock, developers would have to choose one of the many possible implementations of the interface, making the test more coupled to the specific implementation. The use of mock objects can also drive developers towards a better design: Our findings show that a class that requires too much mocking could have been better designed to avoid that. Interestingly, the idea of using the feedback of the test code to improve the quality of production code is popular among TDD practitioners [6].

In addition, our study provides evidence that the coupling between the production code being mocked and test code indeed exists and may impact the maintenance of the software system. Around 50% of changes in mock objects are actually due to implementation changes in production classes, *e.g.*, the signature or the return type of a method changed, and the developer had to fix the mock. In practice, this means that developers

are often required to fix their test code (more specifically, the mocks in these test classes) after changing production classes.

The empirical knowledge on the mocking practices of developers can be useful to tool makers as it (1) provides more awareness on how mocks are used as well as on the possibly problematic coupling between test and production code, and it (2) eases the mocking of similar infrastructure-related dependencies.

More specifically, to the former, we see tool makers proposing ways to warn developers about the usage and the impact their mocks (*e.g.*, "this dependency is often mocked" or "X mocks would be affected in this production changes"). Moreover, we raise the question on whether mocking APIs could be done in such a way that the existing (and currently strong) coupling between test and production code would be smaller.

To the latter, we foresee tool makers proposing tools that would help developers in mocking infrastructure-related code (*e.g.*, database access, file reading and writing), as they are the most popular types of mocked dependencies. A developer who mocks a database dependency will likely spend time simulating common actions, such as "list all entities" and "update entity". A tool could spare the time developers spend in creating repeated simulations for similar types of dependencies (*e.g.*, DAOs share many similarities in common).

Research Question 3. What information do developers need to conduct a proper code review?

To answer our third research question, we considered three large open-source software projects and manually analyzed 900 code review discussion threads that started from a reviewer's question, focusing on what kind of questions are asked in these comments and their answers. In addition, we conducted four semi-structured interviews with developers from the considered systems and one focus group with developers of SIG, both to challenge our outcome and to discuss developers' perceptions.

Our analysis led to seven high-level information needs and, among these, understanding whether a proposed alternative solution is valid and whether the reviewer has the correct knowledge on the code under review are not only the most recurring needs, but also those perceived as the most important. When discussing these topics with developers from both open-source and industrial systems, we uncovered possible areas where current code review tools can offer better features. For example, a key need for the reviewers is being able to communicate with the experts of the sub-system under review; this underlines the importance of tools able to recognize developers' expertise and create recommendations. Researchers have conducted the first steps into this direction: For instance, Patanamon *et al.* [290] proposed RevFinder, an approach to search and recommend reviewers based on similarity of previous reviewed files.

Implications. An interesting novelty that emerged from our analysis, with respect to existing previous work on reviewers recommendation, is the *target* of the recommendation. In fact, existing reviewer recommendation mechanisms target the author of the change who has to select the reviewer and propose reviewers for full changes or files. Instead, we found that also the reviewers have the need to consult an external expert, maybe for

a more specific part of the entire change under review. Targeting reviewers instead of change authors and having a finer grained focus for the recommendation mechanism can lead to interesting changes in both the model (which may use different features to compute expertise and the *difference* of expertise among reviewers) and the evaluation approach (which may no longer be based on just matching actually selected reviewers).

As for proposing an alternative solution, a promising avenue for an impactful improvement in code review is to integrate a tool that automatically mines alternative solutions. Accordingly, a first interesting step would be to investigate how to integrate at code review time an approach such as the one proposed by Ponzanelli *et al.*, which systematically mines community based resources such as forums or Stack Overflow to propose related changes [291]. Another promising starting point in this direction is the concept of programming with "Big Code," as proposed by Vechev and Yahav [292], to automatically learn alternative solutions from the large amount of code available in public code repositories such as GitHub.

Research Question 4. Why and how do developers review test code?

With this research question, we sought to understand how test code is reviewed, identifying current practices and revealing the challenges faced during the reviews of test code. To answer this research question, we conducted two studies. In the first study, we analyzed more than 374,071 code reviews related to three open source projects, investigating the differences between reviews performed on test and production code. Then, we interviewed developers that perform code reviews on a daily basis to understand how they review test files and the challenges they face. In the same study, we discovered that some developers perform (what we later called) Test-Driven Review. Hence, in the second study we focus on this practice: first, we study the effects of TDR in terms of the proportion of defects and maintainability issues found in a review. To this aim, we devise and analyze the results of an online experiment in which 92 developers complete 154 reviews. Then, we investigate problems, advantages, and frequency of adoption of TDR, by means of nine interviews with experiment participants and an online survey with 103 respondents.

In these studies we discovered that test files are almost 2 times less likely to be discussed during code review when reviewed together with production files. When reviewing tests, developers mainly discuss better testing practices, tested and untested paths, and assertions. As for the practices they follow, a main concern of reviewers is understanding whether the test covers all the paths of the production code and to ensure tests' maintainability and readability. Furthermore, we discovered a lack of test-specific information within the code review tool: for example, developers often have to check out the code under review and open it in a local IDE, so they can have a full picture of the code and run the test code. As for the challenges, according to the participants, reviewing test code is harder because (1) it requires them to have context about both the test and the production file under test, and (2) test files are long and are often new additions.

As for TDR, we discovered that participants applying this practice find significantly more bugs in test code, yet fewer maintainability issues in production code. As for the perceived problems with TDR, developers report to (1) consider tests as less important than production, (2) not being able to extract enough knowledge from tests, (3) not being

able to start a review from tests of poor quality, and (4) being comfortably used to read the patch as presented by their code review tool. On the other hand, among the main advantages we have that TDR (1) allows developers to have a concise, high-level overview of the code under test and (2) helps them in being more testing-oriented, hence improving the overall test code quality.

Implications. Most reviewers deem reviewing tests as less important than reviewing production code. Especially when inspecting production and test files that are bundled together, reviewers tend to focus more on production code with the risk of missing bugs in the tests.

Even though many books and articles have been written by practitioners on best practices for code review [293–295] and researchers continue to conduct studies to increase our empirical understanding of MCR [28, 180], best practices for reviewing test code have not been in-depth discussed nor proposed yet. This research question highlights *some* of the current practices used by developers when reviewing test files, such as TDR, but it is not enough: researchers on MCR need to focus more on how developers do code review on test code, and their needs, so that we can better support them.

We identified many pitfalls that reviewers are facing that could be solved by the creation of new tools. For example, many developers complained about the lack of context: when reviewing test code, it is important to understand and inspect the classes that are under test as well as which dependencies are simulated by tests (*i.e.*, mock objects). However, knowing which classes are executed by a test normally requires dynamically executing the code during review, which is not always feasible, especially when large code bases are involved [296]. This is an opportunity to adapt and extend existing research that determines the coverage of tests using static analysis [297]. Moreover, developers would like to be able to easily navigate through the test and tested classes; future research studies could investigate how to improve file navigation for code review in general (an existing open research problem [190]) but also to better support review of tests in particular.

Another important task during the review of a test code is to make sure the test covers all the relevant paths of the production code. Although external tools provide code coverage support for developers (*e.g.*, Codecov [1]), this information is usually not "per test method", *i.e.*, coverage reports focus on the final coverage after the execution of the entire test suite, and not for a single test method. Therefore, new methods should be devised to not only provide general information on code coverage, but also provide information that is specific to each test method.

10

Research Question 5. Does availability bias affect the code review outcome?

In the peer review setting for scientific articles, reviewers normally judge the merit of the submitted work independently from each other. In the MCR setting instead, the comments of the reviewers are immediately visible to the others. With this research question we want to understand whether this visibility can influence the judgment of the other reviewers. To this aim, we devised and conducted a controlled experiment to test the current

[1] https://codecov.io

code review setup and reviewers' proneness to availability bias. The experiment was completed by 85 developers, 73% of which reported to have at least three years of professional development experience. Each developer conducted a code review in which an existing comment was either shown (treatment group) or not (control group).

Our results show that—for three out of four bugs—the code review outcome *does not change* between the treatment and control groups. We could find no evidence indicating that, for these three bugs, the outcome of the review is biased in the presence of an existing review comment priming them on a bug type. Only for one bug type, though, we have strong evidence that the behavior of the reviewers changed: When a review comment is about a type of bug that is normally *not* considered during developers' coding/review practices, the other reviewers were more likely to find the same type of bug with a strong effect.

Implications. The results of our experiment indicate that existing review comments do not act as negative *primers*, rather as positive *reminders*. As such, our experiment provides evidence that the current collaborative code review practice, adopted by most software projects, could be more beneficial than separate individual reviews, not only in terms of efficiency and social advantages, but also in terms of its effectiveness in finding bugs.

10.3 Concluding remarks

Let us look back to our thesis:

> **Thesis.** *By devising new techniques and tools, informed by the current practice, to support developers when writing and reviewing test code, we can increase the overall quality of software systems.*

In this dissertation we show that a relation between test code quality and the overall quality of the system exists, and we study current practices and techniques to better write and review test code. This sets the ground and calls for future research effort focused on other aspects and forms of software testing, such as manual testing and static analysis, as well as the interplay between these practices and code quality. For example, static analysis tools are becoming more and more intelligent, and can capture defects even before tests are run. A big advantage of these tools is that they generally run much faster than their counterparts and do not require a complex setup.

An important, forward-looking question to ponder about in the context of this dissertation is the following one. Given that more and more static tools are created to support developers in verifying the code they write does not contain defects, what is the role of testing going to be? Are tests going to be a vestige from the past hidden in some legacy code? We do not think so, and for a couple of reasons. The first reason has to do with the types of defects these tools can find. As previously discussed, by definition most emerging analysis tools do not execute the code: this means that they are often used to *verify* that the product complies with regulations, requirements, or specifications (*e.g.*, are the function's parameters of the right type? Is the function returning the right object?). On the other hand, tests do execute the code, and therefore used to *validate* the product: does

it do what it is supposed to? how well does it do it? how precise is it? To answer these questions developers need to execute and measure different aspects of the code.

The other reason why we believe tests will not be overthrown any time soon is because tests are also written to avoid regressions. A real-world example of this is you take your car to a mechanic to get the air conditioning fixed, and when you get it back, the air conditioning is fixed but now the car headlights no longer works. None of the mechanics created the defect, but when we put the changes together, the car does not work as expected anymore. Automated software tests have this fundamental role, which is also a peculiarity of software systems.

To conclude, we expect developers to write tests for a long, long time, and we need to help them in this practice. As part of the software engineering research community, we might need to stop thinking of software testing as a self-contained, isolated practice, and start consider it as part of a much larger set of tools that developers use on a daily basis to verify and validate their code. Hence, in the future we need to perform studies on the interplays between these tools and software testing, and how to better support developers when it comes to the usage of all these tools.

Bibliography

References

[1] Robert Lee Hotz. Nasa Mars Accident, 1999.

[2] Arie van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. Refactoring Test Code. In *Proc. of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, pages 92–95, 2001.

[3] Joel Spolsky. The Joel Test: 12 Steps to Better Code. https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/, 2000.

[4] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-Testing : Unit Testing with Mock Objects. *Extreme programming examined*, 2001.

[5] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

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

[7] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.

[8] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated Detection of Test Fixture Strategies and Smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331. IEEE, mar 2013.

[9] Michaela Greiler, Andy Zaidman, Arie van Deursen, and M.-A. Storey. Strategies for Avoiding Text Fixture Smells During Software Evolution. In *Proc. of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 387–396. IEEE, 2013.

[10] B Van Rompaey, B Du Bois, S Demeyer, and M Rieger. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Transactions on Software Engineering*, 33(12):800–817, dec 2007.

[11] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? An empirical study. *Empirical Software Engineering*, 20(4):1052–1094, aug 2015.

[12] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, ASE 2016, pages 4–15, New York, New York, USA, 2016. ACM Press.

[13] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. Code Review: Veni, ViDI, Vici. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 151–160. IEEE, 2015.

[14] Richard A Baker Jr. Code reviews enhance software quality. In *Proceedings of the 19th International Conference on Software Engineering*, pages 570–571. ACM, 1997.

[15] Norihito Kitagawa, Hideaki Hata, Akinori Ihara, Kiminao Kogiso, and Kenichi Matsumoto. Code review participation: game theoretical modeling of reviewers in gerrit datasets. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 64–67. ACM, 2016.

[16] Jason Cohen. Modern Code Review. In Andy Oram and Greg Wilson, editors, *Making Software: What Really Works, and Why We Believe It*, chapter 18, pages 329–338. O'Reilly, 2010.

[17] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down. In *Proceedings of the 2015 International Conference on Software Engineering*. IEEE – Institute of Electrical and Electronics Engineers, may 2015.

[18] Fevzi Belli and Radu Crisan. Towards automation of checklist-based code-reviews. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 24–33. IEEE, 1996.

[19] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proc. of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.

[20] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review. In *Proceedings of the 40th International Conference on Software Engineering*, pages 677–687, New York, NY, USA, may 2018. ACM.

[21] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, may 2013.

[22] Ulrike Abelein and Barbara Paech. Understanding the Influence of User Participation and Involvement on System Success: a Systematic Mapping Study. *Empirical Software Engineering*, 20(1):28–81, 2015.

[23] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050. ACM, 2016.

[24] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer Review on Open Source Software Projects: Parameters, Statistical Models, and Theory. *ACM Transactions on Software Engineering and Methodology*, page 34, 2014.

[25] Chris Sauer, D Ross Jeffery, Lesley Land, and Philip Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *Software Engineering, IEEE Transactions on*, 26(1):1–14, 2000.

[26] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, 1998.

[27] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, Saint Petersburg, Russia, 2013. ACM.

[28] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Review participation in modern code review. *Empirical Software Engineering*, 22(2):768–817, apr 2017.

[29] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review. In *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18*, pages 181–190, New York, New York, USA, 2018. ACM, ACM Press.

[30] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What Makes A Code Change Easier To Review? An Empirical Investigation On Code Change Reviewability. In *26th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page forthcoming, 2018.

[31] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion Detection in Code Reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 549–553. IEEE, sep 2017.

[32] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, oct 2016.

[33] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, volume 2015-Augus, pages 168–179. IEEE, may 2015.

[34] Tobias Baum and Kurt Schneider. On the Need for a New Generation of Code Review Tools. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*, pages 301–308. Springer, 2016.

[35] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. On the Optimal Order of Reading Source Code Changes for Review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 329–340. IEEE, sep 2017.

[36] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering*, 2019.

[37] Patanamon Thongtanunam and Ahmed E Hassan. Review Dynamics and Their Impact on Software Quality. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[38] Paul Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media, 2004.

[39] Steve Freeman and Nat Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[40] Roy Osherove. *The Art of Unit Testing: With Examples in .NET*. Manning, 2009.

[41] Tomek Kaczanowski. *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, 2012.

[42] Jeff Langr, Andy Hunt, and Dave Thomas. *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf, 2015.

[43] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[44] BetterCodeHub. https://bettercodehub.com.

[45] Davide Spadini. PyDriller. Appendix. https://doi.org/10.5281/zenodo.1327363, 2017.

[46] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On The Relation of Test Smells to Software Code Quality. Appendix. https://doi.org/10.5281/zenodo.4075336.

[47] Davide Spadini, Martin Schvarcbacher, Ana Oprescu, Magiel Bruntink, and Alberto Bacchelli. Investigating Severity Thresholds for Test Smells. Appendix. https://doi.org/10.5281/zenodo.3611111, 2020.

[48] Davide Spadini, Mauricio Aniche, Magiel Bruntink, and Alberto Bacchelli. To Mock or Not to Mock? Appendix. https://doi.org/10.5281/zenodo.4075346.

[49] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information Needs In Contemporary Code Review. Appendix. https://doi.org/10.5281/zenodo.3698929.

[50] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When Testing Meets Code Review. Appendix. https://doi.org/10.5281/zenodo.4075318.

[51] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-Driven Code Review: An Empirical Study. Appendix. https://doi.org/10.5281/zenodo.4075356, 2019.

[52] Davide Spadini, Gül Çalikli, and Alberto Bacchelli. Primers or Reminders? The Effects of Existing Review Comments on Code Review. Appendix. https://doi.org/10.5281/zenodo.3653856.

[53] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.

[54] Francisco Zigmund Sokol, Mauricio Finavaro Aniche, and Marco Aurélio Gerosa. MetricMiner: Supporting researchers in mining software repositories. *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013*, pages 142–146, 2013.

[55] K K Chaturvedi, V B Sing, and P Singh. Tools in Mining Software Repositories. In *2013 13th International Conference on Computational Science and Its Applications*, pages 89–98, jun 2013.

[56] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In *2008 International Conference on Software Testing, Verification, and Validation*, volume 3, pages 220–229. IEEE, apr 2008.

[57] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. The Scent of a Smell: An Extensive Comparison between Textual and Structural Smells. *IEEE Transactions on Software Engineering*, 2017.

[58] Steffen M Olbrich, Daniela Cruzes, and Dag I K Sjøberg. Are all code smells harmful? {A} study of God Classes and Brain Classes in the evolution of three open source systems. In *26th {IEEE} International Conference on Software Maintenance {(ICSM} 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10, 2010.

[59] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *Proc. of the 30th International Conference on Software Maintenance and Evolution*, pages 101–110, 2014.

[60] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. Will they like this?: Evaluating code contributions with language models. In *Proc. of the 12th Working Conference on Mining Software Repositories*, pages 157–167. IEEE Press, 2015.

[61] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proc. of the 2nd International Workshop on Mining Software Repositories*, 2005.

[62] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. Are popular classes more defect prone? In *International Conference on Fundamental Approaches to Software Engineering*, pages 59–73. Springer, 2010.

[63] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the Impact of Design Flaws on Software Defects. In *Proc. of the 10th International Conference on Quality Software*, pages 23–31, 2010.

[64] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Re-evaluating Method-Level Bug Prediction. In *Proc. of the 25th International Conference on Software Analysis, Evolution, and Reengineering*, pages 592–601, 2018.

[65] Audris Mockus, Roy T Fielding, and James Herbsleb. A case study of open source software development: the Apache server. In *Proc. of the 22nd international conference on Software engineering*, pages 263–272. Acm, 2000.

[66] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proc. of the 10th Working Conference on Mining Software Repositories*, pages 233–236, 2013.

[67] T J McCabe. A Complexity Measure, 1976.

[68] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proc. of the 35th Int'l Conference on Software Engineering*, pages 422–431, 2013.

[69] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE, sep 2018.

[70] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[71] Stuart Reid. The Art of Software Testing, Second edition. Glenford J. Myers. Revised and updated by Tom Badgett and Todd M. Thomas, with Corey Sandler. John Wiley and Sons, New Jersey, U.S.A., 2004. ISBN: 0-471-46912-2, pp 234. *Software Testing, Verification and Reliability*, 15(2):136–137, jun 2005.

[72] George Candea, Stefan Bucur, and Cristian Zamfir. Automated software testing as a service. In *Proc. of the 1st ACM symposium on Cloud computing*, pages 155–160. ACM, 2010.

[73] Stefan Berner, Roland Weber, and Rudolf K Keller. Observations and Lessons Learned from Automated Testing. In *Proc. of the International Conference on Software Engineering*, pages 571–579. ACM, 2005.

[74] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, mar 2019.

[75] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190, New York, NY, USA, aug 2015. ACM.

[76] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[77] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. Automatic Test Smell Detection Using Information Retrieval Techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 311–322. IEEE, sep 2018.

[78] Martin Fowler. Refactoring: Improving the Design of Existing Code. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 256–256. Addison-Wesley Professional, 2002.

[79] 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 Tom Mens and Serge Demeyer, editors, *Software Evolution*, pages 173–202. Springer, 2008.

[80] M Gatrell and S Counsell. The effect of refactoring on change and fault-proneness in commercial C# software. *Science of Computer Programming*, 102(0):44–56, 2015.

[81] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone java interfaces. In *Software Maintenance (ICSM), 2011 27th International Conference on*, pages 303–312. IEEE, 2011.

[82] Barry Boehm, Dieter H Rombach, and Marvin V. Zelkowitz. *Foundations of Empirical Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[83] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, nov 2014.

[84] Foutse Khomh, Massimiliano Di Penta, Yann Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 2012.

[85] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 2017.

[86] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An

empirical study. In *Proc. of the 9th International Workshop on Search-Based Software Testing*, pages 5–14. ACM, 2016.

[87] M Aniche. Repodriller. https://github.com/mauricioaniche/repodriller, 2012.

[88] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of similarity metrics for refactoring detection. In *Proc. of the 8th Working Conference on Mining Software Repositories*, pages 53–62. ACM, 2011.

[89] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, and Others. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[90] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proc.. International Conference on*, pages 23–32. IEEE, 2003.

[91] Sunghun Kim, E.J. Whitehead, and Yi Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, mar 2008.

[92] Y Kamei, E Shihab, B Adams, A E Hassan, A Mockus, A Sinha, and N Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, jun 2013.

[93] Leon Moonen. Generating Robust Parsers Using Island Grammars. In *Proc. of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, page 13, 2001.

[94] Harry M Sneed. Reverse engineering of test cases for selective regression testing. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proc.. Eighth European Conference on*, pages 69–74. IEEE, 2004.

[95] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 209–218. IEEE, 2009.

[96] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168, 2014.

[97] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017.

[98] Haiquan Li, Jinyan Li, Limsoon Wong, Mengling Feng, and Yap-Peng Tan. Relative Risk and Odds Ratio: A Data Mining Perspective. In *Proc. of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 368–377, New York, NY, USA, 2005. ACM.

[99] Foutse Khomh, M. Di Penta, Y. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on software changeability. *École Polytechnique de Montréal, Tech. Rep. EPM-RT-2009-02*, 2009.

[100] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An Exploratory Study on the Relationship between Changes and Refactoring. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185, 2017.

[101] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[102] Fredi A Diaz-Quijano. A simple method for estimating relative risk using logistic regression. *BMC medical research methodology*, 12(1):14, 2012.

[103] Jun Zhang and F Yu Kai. What's the relative risk?: A method of correcting the odds ratio in cohort studies of common outcomes. *Jama*, 280(19):1690–1691, 1998.

[104] Carsten Oliver Schmidt and Thomas Kohlmann. When to use the odds ratio or the relative risk? *International journal of public health*, 53(3):165–167, 2008.

[105] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE software*, 12(4):52–62, 1995.

[106] Yuming Zhou, Hareton Leung, and Baowen Xu. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. *IEEE Transactions on Software Engineering*, 35(5):607–623, 2009.

[107] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80, dec 1945.

[108] David J Sheskin. Handbook of Parametric and Nonparametric Statistical Procedures. *Technometrics*, 46(3):369–370, aug 2004.

[109] Mr Hess and Jd Kromrey. Robust confidence intervals for effect sizes: A comparative study of Cohen'sd and Cliff's delta under non-normality and heterogeneous variances. *Annual Meeting of the American Educational Research Association*, 2004.

[110] David N Card and William W Agresti. Measuring software design complexity. *Journal of Systems and Software*, 8(3):185–197, 1988.

[111] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[112] Arash Vahabzadeh, Ali Mesbah, Amin Milani Fard, and Ali Mesbah. An Empirical Study of Bugs in Test Code. *2015 {IEEE} International Conference on Software Maintenance and Evolution, {ICSME} 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 101–110, 2015.

[113] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. Investigating Severity Thresholds for Test Smells. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 311–321, New York, NY, USA, jun 2020. ACM.

[114] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 56–65, sep 2012.

[115] Dag I.K. K Sjoberg, Aiko Yamashita, Bente C.D. D Anda, Audris Mockus, and Tore Dyba. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, aug 2013.

[116] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. *IEEE International Conference on Software Maintenance, ICSM*, pages 1–10, sep 2010.

[117] Anthony Peruma. *What the smell? An empirical investigation on the distribution and severity of test smells in open source android applications.* Master thesis, Rochester Institute of Technology, 2018.

[118] Bart van Rompaey, Bart du Bois, and Serge Demeyer. Characterizing the Relative Significance of a Test Smell. In *22nd {IEEE} International Conference on Software Maintenance*, pages 391–400. IEEE, sep 2006.

[119] Manuel Breugelmans and Bart Van Rompaey. TestQ : Exploring Structural and Maintenance Characteristics of Unit Test Suites. *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques.*, 2008.

[120] Abdus Satter, Nadia Nahar, and Kazi Sakib. Automatically identifying dead fields in test code by resolving method call and field dependency. In *CEUR Workshop Proceedings*, 2017.

[121] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, {ISSTA} 2014, pages 385–396. ACM, 2014.

[122] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical Test Dependency Detection. In *2018 {IEEE} 11th International Conference on Software Testing, Verification and Validation ({ICST})*, pages 1–11, apr 2018.

[123] Maudlin Kummer. *Categorising Test Smells.* PhD thesis, University of Bern, 2015.

[124] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, pages 193–202, USA, 2019. IBM Corp.

[125] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[126] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection. *International Workshop on Emerging Trends in Software Metrics, WETSoM*, 2015-Augus:44–53, 2015.

[127] Mauricio Aniche, Christoph Treude, Andy Zaidman, Arie Van Deursen, and Marco Aurelio Gerosa. SATT: Tailoring code metric thresholds for different software architectures. In *Proceedings - 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation, SCAM 2016*, pages 41–50, 2016.

[128] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, 2012.

[129] Efthimia Aivaloglou and Felienne Hermans. How kids code and how we know: An exploratory study on the Scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 53–61. ACM, 2016.

[130] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. Assessing diffusion and perception of test smells in scala projects. *IEEE International Working Conference on Mining Software Repositories*, 2019-May:457–467, 2019.

[131] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[132] Wayne W Daniel. *Applied nonparametric statistics*. {PWS}-Kent, apr 1990.

[133] Hazel E. Nelson. A Modified Card Sorting Test Sensitive to Frontal Lobe Defects. *Cortex*, 12(4):313–324, dec 1976.

[134] Sander Meester. *Towards better maintainable software: creating a naming quality model*. Master thesis, University of Amsterdam, aug 2019.

[135] Dag Sjøberg, Jo Hannay, Ove Hansen, V B Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and A C Rekdal. A survey of controlled experiments in software engineering. *Transactions on Software Engineering*, 31:733–753, 2005.

[136] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. Mock objects for testing java systems: Why and how developers use them, and how they evolve. *Empirical Software Engineering*, 2019.

[137] Davide Spadini, Mauricio Aniche, Magiel Bruntink, and Alberto Bacchelli. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 402–412. IEEE, may 2017.

[138] Per Runeson. A survey of unit testing practices. *IEEE Software*, 2006.

[139] E J Weyuker. Testing component-based software: a cautionary tale. *IEEE Software*, 15(5):54–59, 1998.

[140] Hesam Samimi, Rebecca Hicks, Ari Fogel, and Todd Millstein. Declarative mocking. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 246–256, New York, NY, USA, jul 2013. ACM.

[141] Fergus Henderson. Software Engineering at Google. *CoRR*, abs/1702.0, feb 2017.

[142] Shaikh Mostafa and Xiaoyin Wang. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. In *2014 14th International Conference on Quality Software*, pages 127–132. IEEE, oct 2014.

[143] Madhuri R Marri, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. An Empirical Study of Testing File-System-Dependent Software with Mock Objects. *AST*, 9:149–153, 2009.

[144] Martin Fowler, Kent Beck, and David Heinemeier Hansson. Is TDD dead? https://plus.google.com/events/ci2g23mk0lh9too9bgbp3rbut0k, 2014.

[145] Fabio Pereira. Mockists Are Dead. Long Live Classicists. https://www.thoughtworks.com/insights/blog/mockists-are-dead-long-live-classicists, 2014.

[146] Andrea Arcuri, Gordon Fraser, and René Just. Private api access and functional mocking in automated unit test generation. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 126–137. IEEE, 2017.

[147] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 79–90. ACM, 2014.

[148] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246. ACM, 2004.

[149] Davide Spadini, Mauricio Aniche, Alberto Bacchelli, and Magiel Bruntink. MockExtractor. https://github.com/ishepard/mockextractor.

[150] Gordon Rugg and Peter McGeorge. The sorting techniques: A tutorial paper on card sorts, picture sorts and item sorts. *Expert Systems*, 2005.

[151] D Spencer. Card sorting: a definitive guide. http://boxesandarrows.com/card-sorting-a-definitive-guide/, 2004.

[152] Bruce Hanington and Bella Martin. *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012.

[153] Matt Wynne and Aslak Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[154] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[155] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[156] Andy Hunt and Dave Thomas. *Pragmatic unit testing in c# with nunit*. The Pragmatic Programmers, 2004.

[157] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[158] Magiel Bruntink and Arie Van Deursen. Predicting class testability using object-oriented metrics. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 136–145. IEEE, 2004.

[159] Nornadiah Mohd Razali and Yap Bee Wah. Power comparisons of Shapiro-Wilk , Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, 2(1):21–33, 2011.

[160] Anton J Nederhof. Methods of coping with social desirability bias: A review. *European journal of social psychology*, 15(3):263–280, 1985.

[161] Frens Vonken and Andy Zaidman. Refactoring with unit testing: A match made in heaven? In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 29–38. IEEE, 2012.

[162] Kunal Taneja, Yi Zhang, and Tao Xie. MODA: Automated Test Generation for Database Applications via Mock Objects. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 289, New York, New York, USA, 2010. ACM Press.

[163] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[164] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas A Kraft. Automatically documenting unit test cases. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 341–352. IEEE, 2016.

[165] Michael Karlesky, Greg Williams, William Bereza, and Matt Fletcher. Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns. In *Embedded Systems Conference Silicon Valley (San Jose, California) ESC 413, April 2007*. ESC 413, 2007.

[166] Steve Seunghwan Kim. *Mocking embedded hardware for software validation.* PhD thesis, The University of Texas at Austin, 2016.

[167] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information Needs in Contemporary Code Review. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–27, nov 2018.

[168] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. A Faceted Classification Scheme for Change-Based Industrial Code Review Processes. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, Vienna, Austria, 2016. IEEE.

[169] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 358–368. IEEE, may 2015.

[170] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.

[171] Andrew Sutherland and Gina Venolia. Can peer code reviews be exploited for later information needs? In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 259–262. IEEE, 2009.

[172] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 171–180. IEEE, 2015.

[173] Mika V Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *Software Engineering, IEEE Transactions on*, 35(3):430–448, 2009.

[174] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 111–120. IEEE, sep 2015.

[175] Andrew J Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353, 2007.

[176] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information Needs in Bug Reports : Improving Cooperation Between Developers and Users. *Proceedings of the 2010 Computer Supported Cooperative Work Conference*, pages 301–310, 2010.

[177] Raymond P L Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings - International Conference on Software Engineering*, pages 987–996, 2012.

[178] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34. ACM, 2006.

[179] J D Herbsleb and E Kuwana. Preserving knowledge in design projects: What designers need to know. *Chi '93 & Interact '93*, pages 7–14, 1993.

[180] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 192–201, New York, New York, USA, 2014. ACM, ACM Press.

[181] Deborah Finfgeld-Connett. Use of content analysis to conduct knowledge-building and theory-generating qualitative systematic reviews. *Qualitative Research*, 14(3):341–352, 2014.

[182] Klaus Krippendorff. Agreement and information in the reliability of coding. *Communication Methods and Measures*, 5(2):93–112, 2011.

[183] Patricia A. Adler, Peter Adler, and Robert S. Weiss. Learning from Strangers: The Art and Method of Qualitative Interview Studies. *Contemporary Sociology*, 24(3):420, may 1995.

[184] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, may 2015.

[185] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark G J van den Brand, Alexander Serebrenik, Premkumar Devanbu, and Vladimir Filkov. Gender and tenure diversity in GitHub teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3789–3798. ACM, 2015.

[186] Robert K Merton and Patricia L Kendall. The focused interview. *American journal of Sociology*, 51(6):541–557, 1946.

[187] Mike Kuniavsky. *Observing the user experience: a practitioner's guide to user research*. Elsevier, 2003.

[188] W J Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, Inc., 3rd edition, 1999.

[189] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[190] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 2015 International Conference on Software Engineering. IEEE Press*, 2015.

[191] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130. IEEE, may 2013.

[192] Georgios Gousios, Margaret Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings - International Conference on Software Engineering*, 2016.

[193] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, A E Cruz, Kenji Fujiwara, and Hajimu Iida. Who does what during a code review? datasets of oss peer review repositories. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 49–52. IEEE Press, 2013.

[194] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: evaluating contributions through discussion in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–154. ACM, 2014.

[195] Kenji Yamauchi, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Clustering commits for understanding the intents of implementation. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 406–410. IEEE, 2014.

[196] Mart\'\in Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 341–350. IEEE, 2015.

[197] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 180–190. IEEE Press, 2015.

[198] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. The effects of change-decomposition on code review-A Controlled Experiment. *arXiv preprint arXiv:1805.10978*, 2018.

[199] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 931–940. IEEE, may 2013.

[200] Raymond P L Buse and Westley Weimer. Automatically documenting program changes. In *ASE'10 - Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010.

[201] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Ldiff: An enhanced line differencing tool. In *Proceedings of the 31st International Conference on Software Engineering*, pages 595–598. IEEE Computer Society, 2009.

[202] Chris Parnin and Carsten G\"org. Improving change descriptions with change contexts. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 51–60. ACM, 2008.

[203] Luis Fernando Cortes-Coy, Mario Linares-Vasquez, Jairo Aponte, and Denys Poshyvanyk. On Automatically Generating Commit Messages via Summarization of Source Code Changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 275–284. IEEE, sep 2014.

[204] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2002.

[205] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 541, New York, New York, USA, 2008. ACM Press.

[206] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Testing of Software and Communicating Systems*, pages 228–243. Springer, 2007.

[207] Marco di Biase, Magiel Bruntink, and Alberto Bacchelli. A Security Perspective on Code Review: The Case of Chromium. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 21–30. IEEE, oct 2016.

[208] John M. Chambers and Trevor J. Hastie. *Statistical Models in S*. Routledge, nov 2017.

[209] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481. IEEE, nov 2015.

[210] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357, jun 2002.

[211] Truong Ho-Quang, Michel R.V. Chaudron, Ingimar Samuelsson, Joel Hjaltason, Bilal Karasneh, and Hafeez Osman. Automatic classification of UML Class diagrams from images. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2014.

[212] Soni P M . EFFECTIVENESS OF CLASSIFIERS FOR THE CREDIT DATA SET : AN ANALYSIS. *International Journal of Research in Engineering and Technology*, 05(11):78–83, nov 2016.

[213] Megha Aggarwal. Performance analysis of different feature selection methods in intrusion detection. *International Journal of Scientific & Technology Research*, 2013.

[214] Vicki L Creswell JW, Clark P. Designing and Conducting Mixed Methods Research. *Australian and New Zealand Journal of Public Health*, 31(4):388–388, aug 2007.

[215] Eric S. Raymond. The cathedral and the bazaar. *First Monday*, 3(2), mar 1998.

[216] D. L. Parnas and D. M. Weiss. Active design reviews: Principles and practices. *The Journal of Systems and Software*, 1987.

[217] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, jul 2002.

[218] Peter C Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550. ACM, 2011.

[219] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Software*, 29(6):56–61, nov 2012.

[220] Mario Triola. *Elementary Statistics.* Addison-Wesley, 10th edition, 2006.

[221] Johnny Saldaña. The Coding Manual for Qualitative Researchers (No. 14). *Sage*, 2016.

[222] Adrian Furnham. Response bias, social desirability and dissimulation. *Personality and Individual Differences*, 1986.

[223] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-Driven Code Review: An Empirical Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1061–1072. IEEE, may 2019.

[224] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355, Hyderabad, India, 2014. ACM.

[225] Philip Zembrod. Test-Driven Code Review. https://testing.googleblog.com/2010/08/test-driven-code-review.html, 2010.

[226] Steve Freeman. Sustainable Test-Driven Development. https://www.infoq.com/presentations/Sustainable-Test-Driven-Development, 2010.

[227] Victor Basili, Gianluigi Caldiera, Filippo Lanubile, and Forrest Shull. Studies on reading techniques. In *Proc. of the Twenty-First Annual Software Engineering Workshop*, volume 96, page 2, 1996.

[228] Adam A Porter and Lawrence G Votta. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings of the 16th international conference on Software engineering*, pages 103–112. IEEE Computer Society Press, 1994.

[229] Adam A Porter, Lawrence G Votta, and Victor R Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on software Engineering*, 21(6):563–575, 1995.

[230] Adam Porter and Lawrence Votta. Comparing detection methods for software requirements inspections: A replication using professional subjects. *Empirical software engineering*, 3(4):355–379, 1998.

[231] James Miller, Murray Wood, and Marc Roper. Further experiences with scenarios and checklists. *Empirical Software Engineering*, 3(1):37–64, 1998.

[232] Pierfrancesco Fusaro, Filippo Lanubile, and Giuseppe Visaggio. A replicated experiment to assess requirements inspection techniques. *Empirical Software Engineering*, 2(1):39–57, 1997.

[233] Kristian Sandahl, Ola Blomkvist, Joachim Karlsson, Christian Krysander, Mikael Lindvall, and Niclas Ohlsson. An extended replication of an experiment for assessing methods for software requirements inspections. *Empirical Software Engineering*, 3(4):327–354, 1998.

[234] Victor R. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sørumgård, and Marvin V. Zelkowitz. The empirical investigation of Perspective-Based Reading. *Empirical Software Engineering*, 1(2):133–164, 1996.

[235] Yorai Geffen and Shahar Maoz. On Method Ordering. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, 2016.

[236] Benjamin Biegel, Fabian Beck, Willi Hornig, and Stephan Diehl. The Order of Things: How developers sort fields and methods. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 88–97. IEEE, 2012.

[237] Oliver Laitenberger and Jean-Marc DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software*, 50(1):5–31, 2000.

[238] E.F. Weller. Lessons from three years of inspection data (software development). *IEEE Software*, 10(5):38–45, sep 1993.

[239] Filippo Lanubile and Teresa Mallardo. Inspecting automated test code: a preliminary study. In *Agile Processes in Software Engineering and Extreme Programming*, pages 115–122. Springer, 2007.

[240] Oksana Petunova and Solvita Bērziša. Test Case Review Processes in Software Testing. *Information Technology and Management Science*, 20(1), jan 2017.

[241] Dietmar Winkler, Stefan Biffl, and Kevin Faderl. Investigating the temporal behavior of defect detection in software inspection and inspection-based testing. In *International Conference on Product Focused Software Process Improvement*, pages 17–31. Springer, 2010.

[242] Frank Elberzhager, Alla Rosbach, Jürgen Münch, and Robert Eschbach. Inspection and test process integration based on explicit test prioritization strategies. In *Software Quality. Process Automation in Software Development*, pages 181–192. Springer, 2012.

[243] Frank Elberzhager, Jürgen Münch, and Danilo Assmann. Analyzing the relationships between inspections and testing to provide a software testing focus. *Information and Software Technology*, 56(7):793–806, 2014.

[244] Andy Field and Graham Hole. *How to design and report experiments*. Sage, 2002.

[245] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, pages 1–38, 2017.

[246] Ts Flanigan, E McFarlane, and Sarah Cook. Conducting survey research among physicians and other medical professionals: A review of current literature. *Section of the Survey Research Methods ...*, 2008.

[247] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. Factors Influencing Code Review Processes in Industry. In *Proceedings of the ACM SIGSOFT 24th International Symposium on the Foundations of Software Engineering*, Seattle, WA, USA, 2016. ACM.

[248] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[249] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

[250] Gerry Coleman and Rory O'Connor. Using grounded theory to understand software process improvement: A study of Irish software product companies. *Information and Software Technology*, 49(6):654–667, 2007.

[251] Harvey Goldstein, Anthony S. Bryk, and Stephen W. Raudenbush. Hierarchical Linear Models: Applications and Data Analysis Methods. *Journal of the American Statistical Association*, 88(421):386, mar 1993.

[252] G. David Garson. *Logistic Regression: Binary & Multinomial*. Statistical Associates Publishers, 2014.

[253] Parastou Tourani, Bram Adams, and Alexander Serebrenik. Code of conduct in open source projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 24–33. IEEE, feb 2017.

[254] Davide Spadini, Gül Çalikli, and Alberto Bacchelli. Primers or reminders? The Effects of Existing Review Comments on Code Review. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1171–1182, New York, NY, USA, jun 2020. ACM.

[255] Meike Olbrecht and Lutz Bornmann. Panel peer review of grant applications: What do we know from research in social psychology on judgment and decision-making in groups?, 2010.

[256] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodriguez, and Paul Ralph. Cognitive Biases in Software Engineering: A Systematic Mapping Study. *IEEE Transactions on Software Engineering*, 2018.

[257] Karen K. Wollard. Thinking, Fast and Slow. *Development and Learning in Organizations: An International Journal*, 26(4):38–39, jun 2012.

[258] Keith E. Stanovich. What intelligence tests miss: the psychology of rational thought. *Choice Reviews Online*, 46(10):46–5908–46–5908, jun 2009.

[259] Webb Stacy and Jean Macmillan. Cognitive Bias in Software Engineering. *Communications of the ACM*, 1995.

[260] Amos Tversky and Daniel Kahneman. Availability: A heuristic for judging frequency and probability. *Cognitive Psychology*, 1973.

[261] Morton Deutsch and Harold B. Gerard. A study of normative and informational social influences upon individual judgment. *Journal of Abnormal and Social Psychology*, 1955.

[262] Jonathan Baron. *Thinking and Deciding*. Cambridge University Press, Cambridge, 2006.

[263] Mitzi G. Pitts and Glenn J. Browne. Improving requirements elicitation: an empirical investigation of procedural prompts. *Information Systems Journal*, 17(1):89–110, jan 2007.

[264] Radhika Jain, Jaime Muro Flomenbaum, and Kannan Mohan. A cognitive perspective on pair programming. In *Association for Information Systems - 12th Americas Conference On Information Systems, AMCIS 2006*, 2006.

[265] Jeffrey Parsons and Chad Saunders. Cognitive heuristics in software engineering applying and extending anchoring and adjustment to artifact reuse. *IEEE Transactions on Software Engineering*, 30(12):873–888, dec 2004.

[266] T.K. Abdel-Hamid, Kishore Sengupta, and Daniel Ronan. Software project control: an experimental investigation of judgment with fallible information. *IEEE Transactions on Software Engineering*, 19(6):603–612, jun 1993.

[267] Erik Løhre and Magne Jørgensen. Numerical anchors and their strong effects on software development effort estimates. *Journal of Systems and Software*, 116:49–56, jun 2016.

[268] Scott Plous. *The Psyhology of Judgement and Decision Making*. McGraw-Hill, Inc., 1993.

[269] Laura Marie Leventhal, Barbee M. Teasley, Diane S. Rohlman, and Keith Instone. Positive test bias in software testing among professionals: A review. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 210–218. Springer Berlin Heidelberg, 1993.

[270] Gül Çalikli and Ayşe Başar Bener. Influence of confirmation biases of developers on software quality: An empirical study. *Software Quality Journal*, 2013.

[271] Gul Calikli, Ayse Bener, Turgay Aytac, and Ovunc Bozcan. Towards a metric suite proposal to quantify confirmation biases of developers. In *International Symposium on Empirical Software Engineering and Measurement*, 2013.

[272] Gul Calikli and Ayse Bener. Empirical analysis of factors affecting confirmation bias levels of software engineers. *Software Quality Journal*, 2015.

[273] Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 2005.

[274] Hans van Vliet and Antony Tang. Decision making in software architecture. *Journal of Systems and Software*, 2016.

[275] Don A. Moore and Paul J. Healy. The Trouble With Overconfidence. *Psychological Review*, 2008.

[276] Suranjan Chakraborty, Saonee Sarker, and Suprateek Sarker. An exploration into the process of requirements elicitation: A grounded approach, 2010.

[277] Carolyn Mair and Martin Shepperd. Human judgement and software metrics: Vision for the future. In *Proceedings - International Conference on Software Engineering*, 2011.

[278] Adele Gabrielcik and Russell H. Fazio. Priming and Frequency Estimation. *Personality and Social Psychology Bulletin*, 10(1):85–89, mar 1984.

[279] Klaas Andries de Graaf, Peng Liang, Antony Tang, and Hans van Vliet. The impact of prior knowledge on searching in software documentation. In *Proceedings of the 2014 ACM symposium on Document engineering - DocEng '14*, pages 189–198, New York, New York, USA, 2014. ACM Press.

[280] Kannan Mohan and Radhika Jain. Using traceability to mitigate cognitive biases in software development. *Communications of the ACM*, 51(9):110, sep 2008.

[281] Jason E. Robbins and David F. Redmiles. Software architecture critics in the Argo design environment. *Knowledge-Based Systems*, 1998.

[282] Magne Jørgensen and Dag I K Sjoberg. The Importance of NOT Learning From Experience. In *EuroSPI'2000 -European Software Process Improvement*, 2000.

[283] Baeldung. Avoid Check for Null Statement in Java. https://www.baeldung.com/java-avoid-null-check, 2018.

[284] Robert Brautigam. Why I Never Null-Check Parameters. https://dzone.com/articles/why-i-never-null-check-parameters, 2018.

[285] Scott Shipp. Better Null-Checking in Java. https://dev.to/scottshipp/better-null-checking-in-java-ngk, 2019.

[286] Roger Kirk. *Experimental Design: Procedures for the Behavioral Sciences*. SAGE Publications, Inc., 2455 Teller Road, Thousand Oaks California 91320 United States, 2013.

[287] Austin Lee Nichols and Jon K. Maner. The good-subject effect: Investigating participant demand characteristics. *Journal of General Psychology*, 2008.

[288] Alex Zhitnitsky. *The complete guide to Solving Java Application Errors in Production*. OverOps, 2016.

[289] Eric Neumayer and Thomas Plumper. *Robustness Tests for Quantitative Research*. Cambridge University Press, Cambridge, 2017.

[290] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 141–150. IEEE, 2015.

[291] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Supporting Software Developers with a Holistic Recommender System. In *Proceedings of ICSE 2017 (39th ACM/IEEE International Conference on Software Engineering)*. to be published, 2017.

[292] Martin Vechev, Eran Yahav, and Others. Programming with "Big Code". *Foundations and Trends® in Programming Languages*, 3(4):231–284, 2016.

[293] Josh Poley. Best Practices: Code Reviews. https://msdn.microsoft.com/en-us/library/bb871031.aspx.

[294] SmartBear. Best Practices for Code Review. https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/.

[295] Modern Code Review. https://www.slideshare.net/excellaco/modern-code-review.

[296] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice – Experiences from Windows. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 357–366. IEEE, mar 2011.

[297] Tiago L. Alves and Joost Visser. Static estimation of test coverage. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009*, 2009.

Curriculum Vitæ

Davide Spadini

Education

06/2016 – 06/2020	Ph.D. Student, Software Engineering Research Group, Delft University of Technology, The Netherlands, *Dissertation:* Supporting Quality In Test Code For Higher Quality Software Systems, *Promotors*: Prof. Dr. Alberto Bacchelli and Prof. Dr. Arie van Deursen, *Industry Advisor*: Dr. Magiel Bruntink
2013 – 2016	M.Sc. in Computer Science, University of Trento, Italy *Thesis:* The Wormhole Peer Sampling Service implemented over WebRTC *Advisor:* Prof. Dr. Alberto Montresor
2009 – 2013	B.Sc. in Computer Science, Università di Verona, Italy *Thesis:* Project Photos Localization
2005 – 2009	High School in Computer Science, Guglielmo Marconi, Verona, Italy

Experience

06/2020 – 08/2020	Software Engineer Intern, Facebook UK Limited, London
06/2016 – 06/2020	PhD Researcher, Software Improvement Group, The Netherlands
06/2017 – 08/2017	Visiting Researcher, University of Victoria, Canada
01/2014 – 01/2015	Websites Administrator,

	University of Trento, Italy
04/2013 – 09/2013	Informatics Teacher, Scuola Easy, Italy
2011 – 2013	Web Developer, ZeroSeiPlanet
2007 – 2008	Technical Support Office Intern, VOLKSWAGEN GROUP ITALIA S.P.A.

Open Source Software Projects

Pydriller	Python Framework for Git, https://github.com/ishepard/pydriller/

List of Publications

1. *Davide Spadini*, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, Alberto Bacchelli: Investigating Severity Thresholds for Test Smells. Published in the Proceedings of the 17th International Conference on Mining Software Repositories (MSR). Seoul, Republic of Korea, 2020. Acceptance Rate 29.7% (41/138)

2. *Davide Spadini*, Gul Calikli, Alberto Bacchelli: Primers or Reminders? The Effects of Existing Review Comments on Code Review. Published in the Proceedings of the 42nd International Conference on Software Engineering (ICSE). Seoul, Republic of Korea, 2020. Acceptance Rate 20.9% (129/617)

3. *Davide Spadini*, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, Alberto Bacchelli: Test-Driven Code Review: An Empirical Study. Published in the Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE). Montreal, Canada, 2019. Acceptance Rate 20.6% (109/529)

4. *Davide Spadini*, Mauricio Aniche, Magiel Bruntink, Alberto Bacchelli: Mock objects for testing java systems: Why and how developers use them, and how they evolve. Pubished in the Empirical Software Engineering Journal (EMSE).

5. Luca Pascarella, *Davide Spadini*, Fabio Palomba, Magiel Bruntink, Alberto Bacchelli: Information Needs in Contemporary Code Review. Published in the Proceedings of the 21st ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW). Jersey City, USA, 2018. Acceptance Rate 40.8% (295/722)

6. *Davide Spadini*: Practices and Tools for Better Software Testing. Published in the Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) – Doctoral Symposium. Lake Buena Vista, FL, USA, 2018.

7. *Davide Spadini*, Mauricio Aniche, Alberto Bacchelli: PyDriller: Python Framework for Mining Software Repositories. Published in the Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) – Tool Demonstration. Lake Buena Vista, FL, USA, 2018. Acceptance Rate 38.8% (14/36)

8. Hennie Huijgens, *Davide Spadini*, Dick Stevens, Niels Visser, Arie van Deursen: Software Analytics in Continuous Delivery: A Case Study on Success Factors. Published in the Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement (ESEM). Oulu, Finland, 2018. Acceptance Rate 18.2% (30/164)

9. *Davide Spadini*, Fabio Palomba, Andy Zaidman, Magiel Bruntink, Alberto Bacchelli: On The Relation of Test Smells to Software Code Quality. Published in the Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME). Madrid, Spain, 2018. Acceptance Rate 21.2% (37/174)

10. *Davide Spadini*, Mauricio Aniche, Margaret-Anne Storey, Magiel Bruntink, Alberto Bacchelli: When Testing Meets Code Review: Why and How Developers Review Tests. Published in the Proceedings of the 40th International Conference on Software Engineering (ICSE). Gothenburg, Sweden, 2018. Acceptance Rate 20.9% (105/502)

11. *Davide Spadini*, Mauricio Aniche, Magiel Bruntink, Alberto Bacchelli: To Mock or Not To Mock? An Empirical Study on Mocking Practices. Published in the Proceedings of the 14th International Conference on Mining Software Repositories (MSR). Buenos Aires, Argentina, 2017. Acceptance Rate 30.6% (37/121)

Included in this thesis.
Won a Honorable Mention Award or Distinguished Artifact Award