UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Victor Guerra Veloso

Assessing How Developers Review Tests on GitHub

Belo Horizonte
2022

Victor Guerra Veloso

Assessing How Developers Review Tests on GitHub

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Andre Cavalcante Hora

Belo Horizonte
2022

Victor Guerra Veloso

Avaliando Como Desenvolvedores Revisam Testes no GitHub

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Andre Cavalcante Hora

Belo Horizonte
2022

Veloso, Victor Guerra.

V443a Assessing how developers review tests on GitHub [manuscrito] / Victor Guerra Veloso. — 2022.
72 f. il.; 29 cm.

Orientador: André Cavalcante Hora.
Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação
Referências: f. 62-72.

1. Computação – Teses. 2. Engenharia de software – Teses. 3. Software – Testes - Teses. 4. Software – Controle de qualidade – Teses. I. Hora, André Cavalcante. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irenquer Vismeg Lucas Cruz CRB 6ª Região – 819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

ASSESSING HOW DEVELOPERS REVIEW TESTS ON GITHUB

VICTOR GUERRA VELOSO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. André Cavalcante Hora - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Marco Túlio de Oliveira Valente
Departamento de Ciência da Computação - UFMG

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 13 de julho de 2022.

To the unprivileged ones, to whom the world turned its back.
To Fadoa – my love and my study partner

Acknowledgments

I would like to thank the following people, without whom I would not have been able to complete this research, and without whom I would not have made it through my master's degree! The ASERG team at the Federal University of Minas Gerais, especially my supervisor Professor Andre Hora, whose knowledge, patience, and tips I will keep close to my heart forever. And special thanks to Professor Marco Túlio and the masterminds supervised by him for having me in all those Thursday meetings, the recommendations you provided and the opportunity to discuss such valuable topics are immeasurable. Thank you Mario Henrique Mesquita for being a friend, a student, a translator, and a therapist, all at the same time. And my biggest thanks to my family for all the support you have shown me through this research, especially my Mother who taught me the value of doing my best without comparing myself to my pairs. And for my wife, Fadoa, thanks for all your support, without which I would have stopped these studies a long time ago. You have been amazing, and I promise I will start packing up my baggage for our travel soon!

"*Sharing knowledge is the most fundamental act of friendship. Because it is a way you can give something without loosing something.*"

(Richard Stallman)

Resumo

No desenvolvimento de software moderno, espera-se que desenvolvedores forneçam testes adequados para cobrir suas alterações de código. No entanto, contribuições nem sempre incluem bons testes. Dessa forma, os revisores podem solicitar melhoria de testes nas contribuições. Na prática, não está claro quais informações estão disponíveis para orientar os colaboradores na implementação dos métodos de testes solicitados durante a revisão de código. Portanto, entender melhor as práticas de revisão de testes seria importante para orientar tanto os colaboradores quanto os revisores. Nesta dissertação, propomos dois estudos. Primeiro, fornecemos um estudo empírico para avaliar as revisões de testes no GitHub. Encontramos 11.836 avaliações de teste em 5.421 projetos de código aberto, sugerindo que essa é uma prática comum. Também detectamos oito amplas categorias de recomendações em revisões de testes: *escopo do teste*, *suporte a ferramentas*, *cenários de teste*, *objetivo/propósito*, *refatorações*, *más práticas*, *fixtures* e *miscellaneous*. Por fim, descobrimos que as revisões de testes com mais recomendações são mais propensas a serem resolvidas. Em nosso segundo estudo, propomos uma ferramenta para avaliar a qualidade de *métodos de testes* individuais por meio de testes de mutação. Esta ferramenta estende uma framework de testes de mutação estado-da-arte para analisar métodos de teste e relatar resultados de mutação a nível de método. Finalmente, com base em nossos resultados, discutimos implicações para pesquisadores e profissionais.

Palavras-chave: Teste de Software, Revisão de código, Mineração de Repositórios de Software, Manutenção de Software, Teste de Mutação, Qualidade de Software.

Abstract

In modern software development, developers are expected to provide proper tests to cover their code changes. However, code contributions are not always attached to good tests. This way, reviewers may request test changes to the contributions. In practice, it is not clear what information is available to guide contributors in implementing the requested test methods during the code review. Therefore, better understanding test review practices would be important to guide both contributors and reviewers. In this dissertation, we propose two studies. First, we provide an empirical study to assess test reviews on GitHub. We find 11,836 test reviews in 5,421 open-source projects, suggesting that this is a common practice. We also detect eight broad categories of recommendations in test reviews: *test scope*, *tool support*, *test scenarios*, *goal/purpose*, *refactoring*, *bad practices*, *fixtures*, and *miscellaneous*. Lastly, we find that test reviews with more recommendations are more likely to be solved. In our second study, we propose a tool to assess the quality of individual *test methods* by relying on mutation testing. This tool extends a state-of-the-art mutation testing framework to analyze test methods and report mutation results at the method level. Finally, based on our results, we discussed implications for researchers and practitioners.

Keywords: Software testing, Code Review, Software repository mining, Software maintenance, Mutation Testing, Code quality.

List of Figures

1.1 Mandatory request for test addition in code review 16

2.1 Code review in RxJS' pull request #4115 22
2.2 Reviewers verdict, checks, and bot's post 22
2.3 Test reviews of different specificity levels 23
2.4 Test reviews with different context sources 24
2.5 Test review about API breaking change 25
2.6 Traditional mutation testing approach 26
2.7 Score computation example in mutation testing inspired by [62] ("TM": Test Method) 27

3.1 Request for tests covering a specific function 32
3.2 Overview of projects', contributors', and reviewers' social metrics 32
3.3 Overview of the study design 33
3.4 Query Validation Steps 34
3.5 Examples of test review 35
3.6 Discussion's content analysis and classification steps 36
3.7 Distribution of test reviews per programming language 38
3.8 Distribution of recommendations in *add tests* 41
3.9 Distribution of recommendations in *change existing tests* 41
3.10 Solved ratio per request category ("c": *change existing tests*; "a": *add test*) . . 47
3.11 Distribution of recommendations among *unsolved* (left) and *solved* (right) test reviews 47

4.1 Overview of our approach 52
4.2 Score computation example in mutation testing inspired by [62] ("TM": Test Method) 53
4.3 Prevalence of test smells 57

List of Tables

3.1 Top repositories . . . 39
3.2 Test request classes, their categories, and frequencies . . . 39
3.3 Recommendations within a test request, their categories, and frequencies . . . 42
3.4 Contributor's feedback, their categories, and frequencies . . . 46

4.1 Projects test quality overview . . . 54
4.2 Metrics overview ($\bar{\eta}$: median; Rnd: Random; N: Negligible; VS: Very Small; S: Small; H: Huge) . . . 56

Contents

1 Introduction 15
- 1.1 Motivation . . . 15
- 1.2 Proposed Work . . . 16
- 1.3 Contributions . . . 18
- 1.4 Publication . . . 18
- 1.5 Outline of the Dissertation . . . 19

2 Background and Related Work 20
- 2.1 Code Review in a Nutshell . . . 20
- 2.2 Modern Code Review . . . 21
- 2.3 Test Review . . . 23
- 2.4 Mutation Testing . . . 25
- 2.5 Related Work . . . 27
- 2.6 Final Remarks . . . 30

3 How Developers Review Tests on GitHub 31
- 3.1 Study Design . . . 31
- 3.2 Results . . . 37
- 3.3 Discussion and Implications . . . 48
- 3.4 Threats to Validity . . . 49
- 3.5 Final Remarks . . . 50

4 A Tool to Characterize Test Method Quality 51
- 4.1 Mutation Testing at Method Level . . . 51
- 4.2 Study Design . . . 53
- 4.3 Results . . . 56
- 4.4 Discussion and Implications . . . 58
- 4.5 Threats to Validity . . . 58
- 4.6 Final Remarks . . . 59

5 Conclusion 60
- 5.1 Overview and Contributions . . . 60
- 5.2 Future Work . . . 62

Chapter 1

Introduction

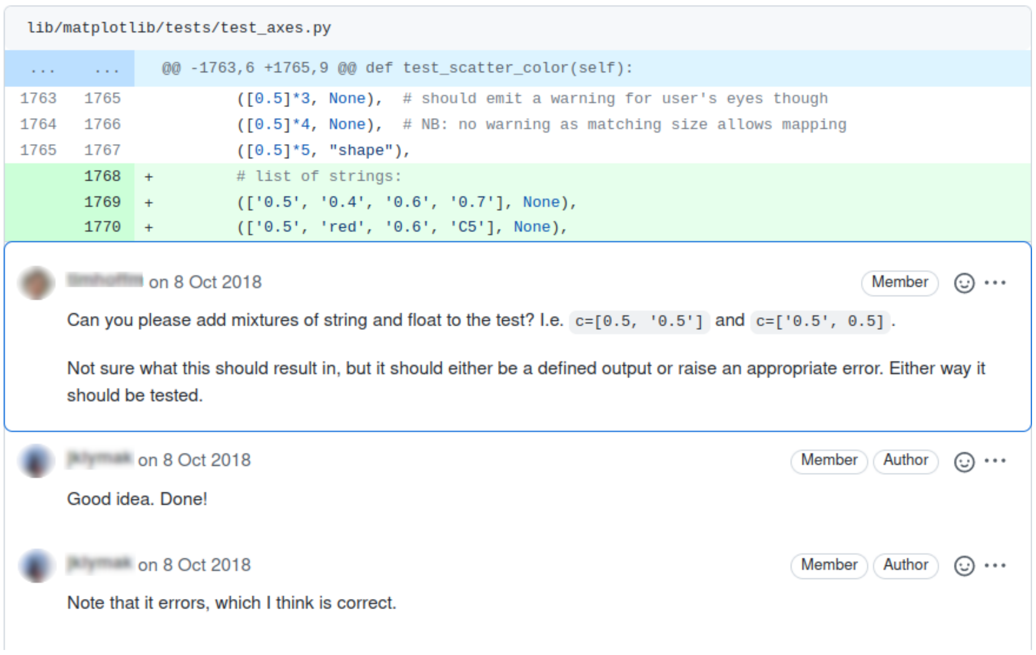
1.1 Motivation

In modern software development, developers are expected to provide proper tests to cover their code changes, like new features and bug fixes [1–3]. Some projects require tests and system-under-test (SUT) changes to be in the same commit,[1] others are more relaxed and just specify test-related goals.[2] Those are two different approaches to addressing the same concern: ensuring the reviewability of a contribution. Reviewability is a desired contribution's quality, especially on open source projects where external contributions are frequent. Thus, higher is the relevance of contributions contextualization [4–7]. The reason is that many projects adopt code review as an essential practice to ensure contribution quality, code familiarity, and style conformance. Thus, an easy-to-review contribution prevents change evaluation latency and speeds up the development of a software project [8–11].

However, code contributions are not always attached to good tests. This way, reviewers may request test changes to the contributions. For instance, Figure 1.1 shows a code review in project matplotlib,[3] whose reviewer asked for a specific test improvement. Detecting uncovered lines in code contributions involves the commitment of the reviewers and even code execution. Fortunately, there are some tools that provide useful information for reviewers, including code coverage reports, and consequently, make the reviewing process easier. These tools are possible due to CI/CD workflows automatically running tests, static analysis tools, and code coverage reporters checks to ensure that the tests are provided or that they have a certain level of coverage [11–14].

In practice, it is not clear what information is available to guide contributors in implementing the requested test methods during the code review. On the one hand, all required information should be provided to the contributor by the reviewer who asked for such tests. On the other hand, some contributors may also get upset with vague, albeit

[1]For example, groupon/assertive ether/etherpad-lite, and bamthomas/aioimaplib
[2]solidusio/solidus ansible-collections/community.general, and cnrancher/octopus
[3]matplotlib/matplotlib#12431



```
lib/matplotlib/tests/test_axes.py
@@ -1763,6 +1765,9 @@ def test_scatter_color(self):
          ([0.5]*3, None),  # should emit a warning for user's eyes though
          ([0.5]*4, None),  # NB: no warning as matching size allows mapping
          ([0.5]*5, "shape"),
+         # list of strings:
+         (['0.5', '0.4', '0.6', '0.7'], None),
+         (['0.5', 'red', '0.6', 'C5'], None),
```

[redacted] on 8 Oct 2018 Member

Can you please add mixtures of string and float to the test? I.e. `c=[0.5, '0.5']` and `c=['0.5', 0.5]`.

Not sure what this should result in, but it should either be a defined output or raise an appropriate error. Either way it should be tested.

[redacted] on 8 Oct 2018 Member Author

Good idea. Done!

[redacted] on 8 Oct 2018 Member Author

Note that it errors, which I think is correct.

Figure 1.1: Mandatory request for test addition in code review

excessive, test reviews: "*Most of them already have some tests or are pretty well exercised (...) So unless you have a specific test case in mind, I won't add any.*"[4] Therefore, better understanding test review practices would be important to guide both contributors and reviewers when discussing the test changes to be implemented.

1.2 Proposed Work

In this dissertation, we propose an empirical study to assess test reviews on GitHub. Specifically, we focus on test reviews that the reviewers ask the contributors to add or change tests. We rely on the GHTorrent [15] dataset and analyze 11,836 test reviews. We also manually analyze a sample 324 test reviews to better understand their content. To support this study, we propose four research questions:

- **RQ1 (overview): How frequently do reviewers ask contributors for tests?** In this research question, we search for test reviews in the GHTorrent in which the reviewer asks for test addition or modification. We find 11,836 test reviews in 5,421 open-source projects, suggesting that this is a common practice.
- **RQ2 (test requests): What do reviewers ask contributors for?** We assess the reviewers' request content and identify two major categories. We find that *add*

[4] iTowns/itowns#792

test is prevalent with 83.7%, while *change existing test*, only represents 16.3% of the collected test reviews. In addition, we find sub-categories, including *fix test*, *refactor test*, and *improve test*.

- **RQ3 (request recommendations): Which recommendations are provided in test requests?** We manually analyzed the reviewers' request to assess their recommendations. We find seven broad categories of recommendations: *test scope*, *tool support*, *test scenarios*, *goal/purpose*, *refactoring*, *bad practices*, *fixtures*, and *miscellaneous*. Those broad categories are composed of 36 specific recommendations that the contributors should focus to improve their tests, such as *e2e test*, *parameterized test*, *test double*, *edge case*, *setup*, to name a few.

- **RQ4 (test responses): How are the test reviews solved by contributors?** Finally, we explore the contributors' response to assess the solved and unsolved test reviews. Overall, we find that test reviews are likely to be solved. We find that contributors solve test *fixes* and *refactoring* more frequently than other changes. Moreover, reviews with more recommendations are more likely to be *solved*.

Overall, our results show that tests may change during test review to accommodate the feedback of the community. Frequently, this may occur after a long discussion in the test review process. After all, it is not clear how good are the implemented test methods, thus, we are left unsure about their final quality.

To shed some light on this direction, we propose a tool to assess the quality of individual *test methods* by relying on mutation testing. This tool extends a state-of-the-art mutation testing framework [16] to analyze test methods and report mutation results at the method level. We assess 18,321 test methods provided by five popular open-source projects: RxJava, OkHttp, Retrofit, ZXing, and Apache Commons Lang. We then propose research questions to assess test method quality:

- **RQ5 (test quality): What are the code and evolutionary characteristics of high-quality test methods?** In this question, we rank test methods by mutation score and extract the top-100 test methods comprising the *best* test methods group and the bottom-100 to build the *worst* test methods group. We compute seven metrics for each group and find no major difference between them.

- **RQ6 (test smells): What test smells are prevalent in high-quality test methods?** In this final question, we detect the prevalent *test smells* in each group from the previous RQ, *i.e., best* and *worst*. We find the *worst* test methods are overconcentrated on critical test smells, while the *best* group is likely to contain test smells related to maintainability.

1.3 Contributions

The contributions of this research are sixfold:

- We provide empirical evidence that test is an essential part of a contribution to open-source projects and reviewers request for test addition in case they perceive insufficient code coverage.
- We provide a set of 36 recommendations extracted from test requests, each thoroughly discussed and demonstrated with several real-world examples.
- We propose a tool to assess mutation testing for test methods individually.
- We provide a dataset [17] of 11,836 test reviews from 5,421 open-source projects, of which 373 were manually evaluated, categorized, and linked to their solving patch if applicable. This can be used to build and improve existing tools for automatic linkage [18–20].
- We provide a dataset [21] of 15,970 test methods from five popular open-source projects with 5 dynamic quality metrics, 34 static quality metrics, and 7 evolution metrics each.
- We discuss implications for practitioners and researchers working on software testing, and implications for reviewers and contributors working on test review.

1.4 Publication

This master dissertation produced the following publication, and, therefore, it contains its material:

- Victor Veloso, Andre Hora. Characterizing High-Quality Test Methods: A First Empirical Study. In *19th IEEE/ACM International Conference on Mining Software Repositories* (MSR), pages 1-5, 2022.

1.5 Outline of the Dissertation

The remaining of this dissertation is organized as follows:

- **Chapter 2** presents the related work and explains the related fundamental concepts. Then, we present Modern Code Review, its particularities, and associated tooling and techniques. Next, we introduce the test review concept, the main concept this dissertation is built upon, and exhibit some real-world examples of its usage. Finally, we characterize mutation testing, a powerful test quality measurement technique, and demonstrate some of its limitations.

- **Chapter 3** describes our empirical study, in which we assess code reviews taking place on the GitHub platform that are indexed by the GHTorrent dataset. First, we detail the study design and four research questions. Next, we reveal the results. Lastly, we present discussion, implications, and threats to validity.

- **Chapter 4** introduces the tool we designed to measure the quality of test methods individually. We detail the design of our technique and present its advantages through an example. Then, we proceed to the study design of an exploratory study focused on showing how this technique relates to many other static-analysis metrics. Next, we present the results of our two extra research questions. Finally, we discuss the results, implications, and threats to validity.

- **Chapter 5** concludes this master dissertation by covering the overview of our findings, the main contributions, and the intended future work.

Chapter 2

Background and Related Work

This chapter provides the background for this dissertation. First, in Section 2.1, we introduce some aspects of code review. Second, in Section 2.2, we discuss how code review evolved to Modern Code Review. Next, in Section 2.3, we present test review and examples from GitHub. Finally, in Section 2.4, we introduce mutation testing, which is one of the most thorough approaches to measure test quality.

2.1 Code Review in a Nutshell

Software development is a collaborative task and, as such, it involves code familiarity, level of agreement, team effectiveness [22], design and style conformance [23–29], and contribution quality assurance [30–36]. In this context, code review is a practice used by many companies [37–40] and communities [7, 35, 36, 41, 42] to promote all or some of those goals. It includes two roles [43, 44]: the contributor and the reviewer, also called integrator by some authors [45]. The *contributor*, who is the author of the code being reviewed, answers the review questions and fixes the identified issues. The *reviewer* identifies whether the contributed code contains bugs, design problems, mismatched style, or bad quality. Next, the reviewer can reject, accept, or even conditionally accept a contribution, requesting changes on faulty fragments of code. Traditionally, code review was based on email exchange [46, 47], then on specialized software. Afterwards, the code review practice evolved to Modern Code Review [48], following the fast-pacing and dynamic flow of modern software development.

2.2 Modern Code Review

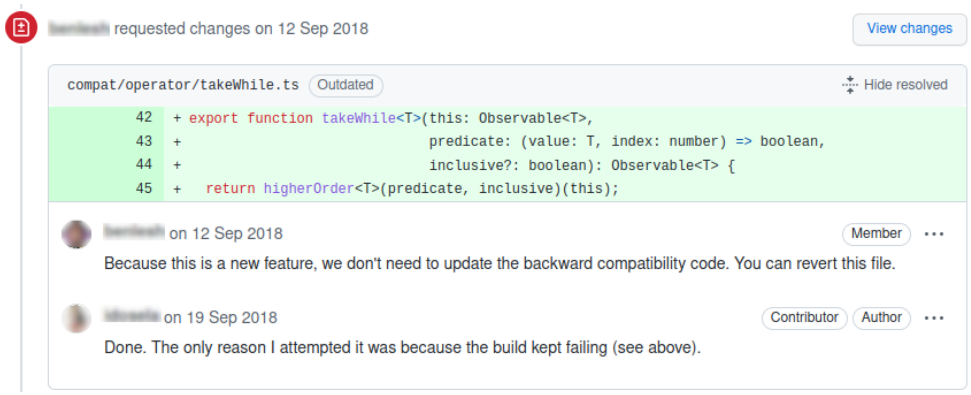
By definition, Modern Code Review combines four characteristics: informal, asynchronous, tool-based, and frequent [48, 49]. The adoption of informality replaces inspection's inconvenience by lack of traceability, which can be further mitigated by introducing specialized tooling that assists reviewers [49]. Being asynchronous, Modern Code Review allows reviewers to prioritize the contributions with which they are more familiarized and more interested, thus increasing review productivity. Tools also help to deal with the asynchronous aspects of the Modern Code Review process, by assisting developers to explore contributions and follow a structured review flow. For instance, Google has its own modern code review tool, called `CRITIQUE`, that supports a five-step flow similar to those usually found in tool-based reviews: creating, previewing, commenting, addressing feedback, and approving [37]. Modern Code Review is *frequent*, which means it is fast, continuous, and "*often has more similarities with pair programming than with inspection*" [49]. Besides, it is commonly advised to have multiple reviewers for each contribution [33, 34]. In those cases, the final acceptance is conditioned upon the consensus of all assigned reviewers. In addition, some projects set up bots [50, 51] to act as reviewers, judging whether the contribution should be accepted, by running the test suite [11–14, 52] and static analysis [53, 54].

As evidenced by the literature [55, 56], most projects hosted on GitHub adopt the Pull-based development model. This model brings communication to a higher level of contribution granularity, thus it enables visualizing the contribution's big picture [56]. In this context, studies investigated pull requests and their interested parties' expectations and perspectives, and found analogies to other tool-based code reviews [45, 57]. The pull-based development model is also known as a Modern Code Review practice that takes place on GitHub repositories, specifically, in their pull requests section [37]. Currently, GitHub supports four types of comments: line comments, review comments, issue comments, and commit comments. The first two are specific to pull requests because they are pinned to a line change of the contribution. While line comment is the simplest, review comments are tied to the code review verdict of one reviewer. For instance, a review comment can be either associated with an acceptance, a rejection, or a conditional acceptance (*i.e.,* a request for change). Commit comment, as the name suggests, belongs to a single commit in the commit history. Issue comment is, despite the name, supported by both issues and pull requests.

Figure 2.1 presents an example of Modern Code Review from an open-source project hosted on GitHub.[1] The specific pull request has two reviewers (*benlesh* and

[1] ReactiveX/rxjs#4115

cartant) and three bots (AppVeyor, Travis CI, and coveralls). Given the project's guidelines, a pull request may require a consensus between all reviewers and bots. That means the reviewers must be satisfied as well as the contribution's tests must pass on Windows, pass on Linux, and at least maintain the code coverage rate.


[redacted] requested changes on 12 Sep 2018

View changes

compat/operator/takeWhile.ts Outdated

Hide resolved

```
+ export function takeWhile<T>(this: Observable<T>,
+                              predicate: (value: T, index: number) => boolean,
+                              inclusive?: boolean): Observable<T> {
+   return higherOrder<T>(predicate, inclusive)(this);
```

[redacted] on 12 Sep 2018 Member

Because this is a new feature, we don't need to update the backward compatibility code. You can revert this file.

[redacted] on 19 Sep 2018 Contributor Author

Done. The only reason I attempted it was because the build kept failing (see above).

Figure 2.1: Code review in RxJS' pull request #4115

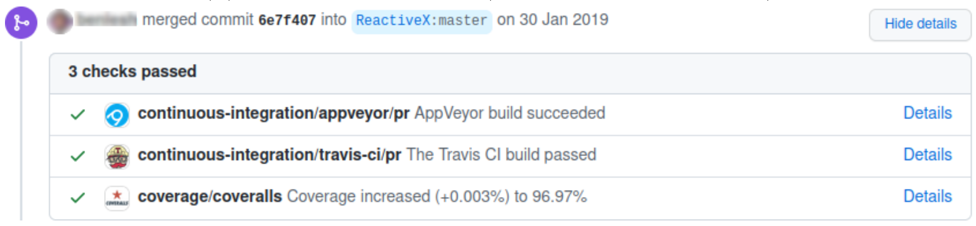
Next, Figure 2.2a presents the reviewers' verdict and Figure 2.2b displays the details of checks made by bots for the merge commit of that pull request. Lastly, Figure 2.2c shows an example of a coverage report posted by a bot.

Reviewers

[redacted] ✓

[redacted] ✓

(a) Reviewers verdict (Pull request accepted)


[redacted] merged commit `6e7f407` into `ReactiveX:master` on 30 Jan 2019

Hide details

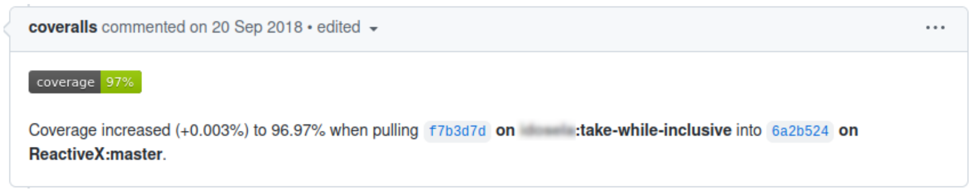
3 checks passed

✓ **continuous-integration/appveyor/pr** AppVeyor build succeeded Details

✓ **continuous-integration/travis-ci/pr** The Travis CI build passed Details

✓ **coverage/coveralls** Coverage increased (+0.003%) to 96.97% Details

(b) Bots' checks report (All checks passed)


coveralls commented on 20 Sep 2018 • edited

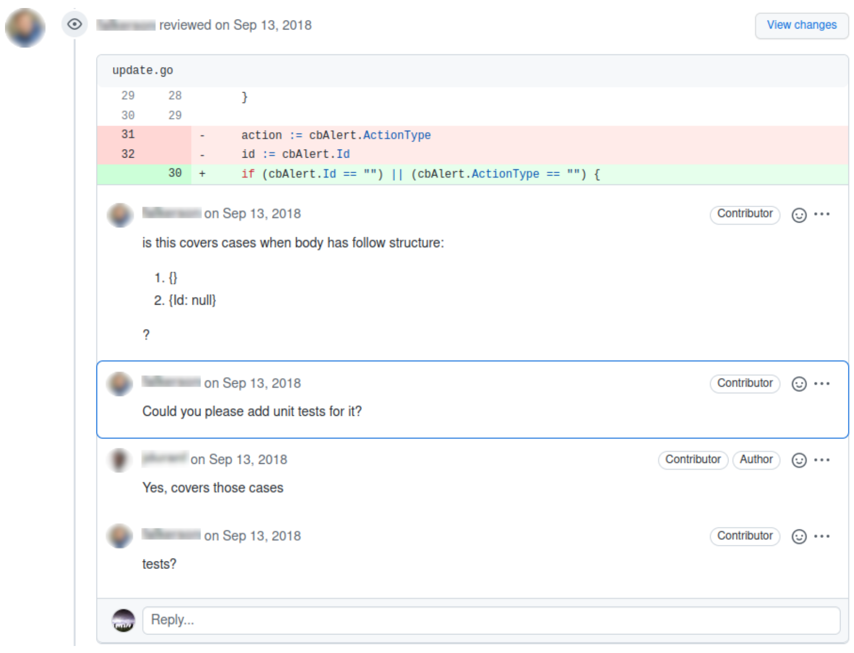
coverage 97%

Coverage increased (+0.003%) to 96.97% when pulling `f7b3d7d` **on** [redacted]**:take-while-inclusive** into `6a2b524` **on ReactiveX:master**.

(c) Coveralls bot's post about code coverage of the pull request

Figure 2.2: Reviewers verdict, checks, and bot's post

2.3 Test Review



[redacted] reviewed on Sep 13, 2018 View changes

update.go

```
29   28        }
30   29
31        -    action := cbAlert.ActionType
32        -    id := cbAlert.Id
     30   +    if (cbAlert.Id == "") || (cbAlert.ActionType == "") {
```

[redacted] on Sep 13, 2018 Contributor

is this covers cases when body has follow structure:

1. {}
2. {Id: null}

?

[redacted] on Sep 13, 2018 Contributor

Could you please add unit tests for it?

[redacted] on Sep 13, 2018 Contributor Author

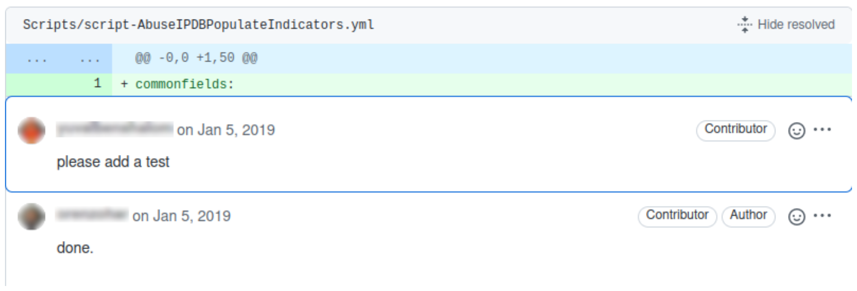
Yes, covers those cases

[redacted] on Sep 13, 2018 Contributor

tests?

Reply...

(a) Test review to cover specific cases



Scripts/script-AbuseIPDBPopulateIndicators.yml Hide resolved

```
...  ...  @@ -0,0 +1,50 @@
       1  + commonfields:
```

[redacted] on Jan 5, 2019 Contributor

please add a test

[redacted] on Jan 5, 2019 Contributor Author

done.

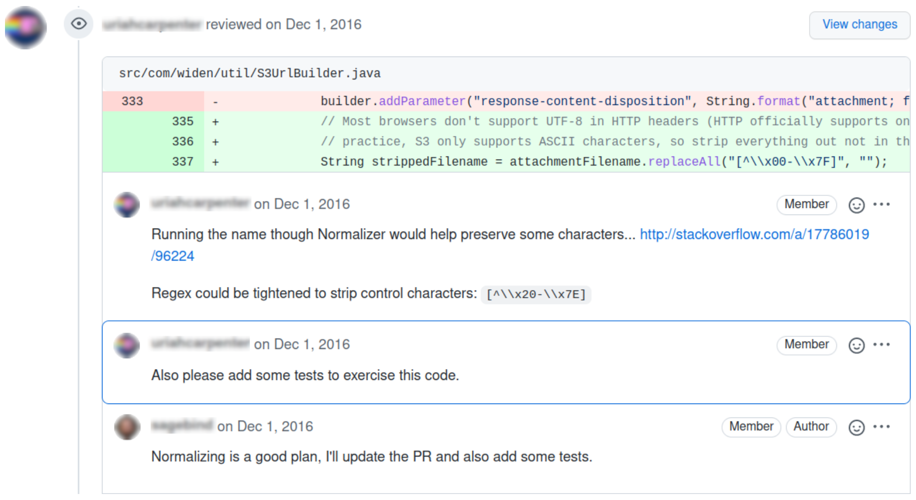
(b) Vague test review

Figure 2.3: Test reviews of different specificity levels

Code review can target either production code, test code, or both. At Google, code review introduction was explained by three benefits: "*checking the consistency of style and design; ensuring adequate tests; and improving security.*" [37] Indeed, that perceived relationship between code review and test code emerges as a research line [58]. The more automated tests become ubiquitous, the more compelling the ability to distinguish general code review and test-specific code review, especially in research. Therefore, we call hereafter code reviews targeting test code as test reviews.

A *test review* may address uncovered code of different granularities or even not mention any SUT. Next, we compare two examples of test reviews that are opposites in their specificity. The first example,[2] in Figure 2.3a, shows a test review whose reviewer pinned a change in line 30 of the `update.go` file, enumerated two inputs, asked whether

[2] edgexfoundry/device-sdk-go#51



[redacted] reviewed on Dec 1, 2016

src/com/widen/util/S3UrlBuilder.java

```
-            builder.addParameter("response-content-disposition", String.format("attachment; f
+            // Most browsers don't support UTF-8 in HTTP headers (HTTP officially supports on
+            // practice, S3 only supports ASCII characters, so strip everything out not in th
+            String strippedFilename = attachmentFilename.replaceAll("[^\\x00-\\x7F]", "");
```

[redacted] on Dec 1, 2016 · Member

Running the name though Normalizer would help preserve some characters... http://stackoverflow.com/a/17786019/96224

Regex could be tightened to strip control characters: `[^\\x20-\\x7E]`

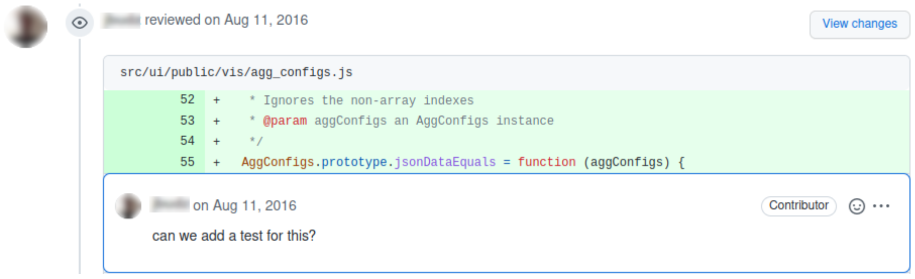
[redacted] on Dec 1, 2016 · Member

Also please add some tests to exercise this code.

[redacted] on Dec 1, 2016 · Member · Author

Normalizing is a good plan, I'll update the PR and also add some tests.

(a) Context from a previous discussion



[redacted] reviewed on Aug 11, 2016

src/ui/public/vis/agg_configs.js

```
+    * Ignores the non-array indexes
+    * @param aggConfigs an AggConfigs instance
+    */
+   AggConfigs.prototype.jsonDataEquals = function (aggConfigs) {
```

[redacted] on Aug 11, 2016 · Contributor

can we add a test for this?

(b) Context is the pinned line of code

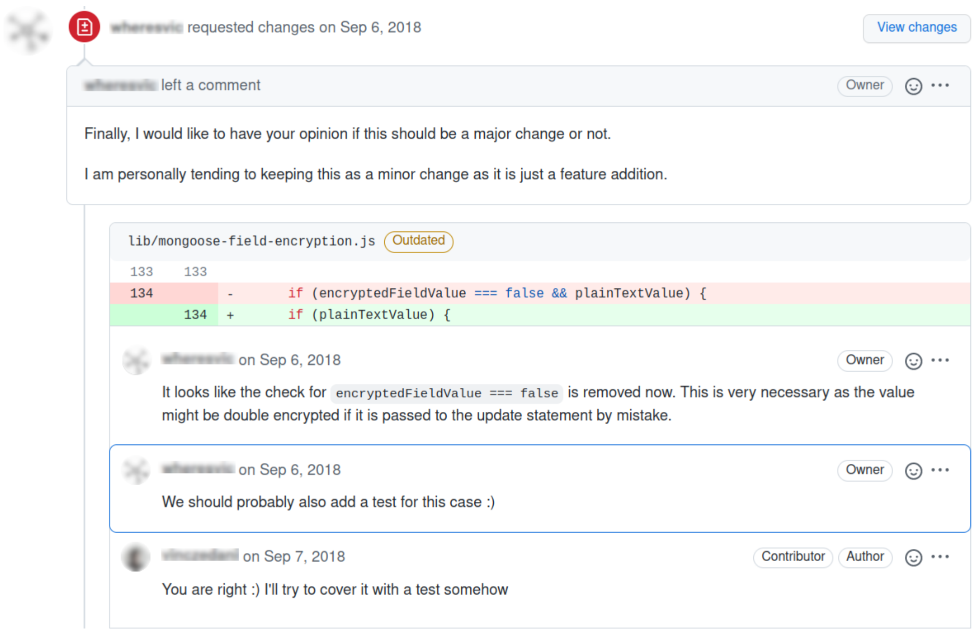
Figure 2.4: Test reviews with different context sources

they are properly handled by the production code, and requested the contributor to add test cases in the test suite to exercise such scenarios. In contrast, Figure 2.3b shows an example[3] of a vague test review, in which the reviewer pinned the first line of an YAML file and requested for test addition, but mentioned neither the input nor the SUT.

Furthermore, test reviews may have different context sources or even a combination of multiple sources (*i.e.*, the line and name of the pinned file, previous discussions, other issues/pull requests/commits, names of code structures, and code comments). The context plays an important role in one's ability (or inability) to understand what is being requested, hence sudden context can lead to confusion [59–61]. For example, Figure 2.4a shows a test review whose discussion is self-explanatory, whereas it still has links to external resources. On the other hand, the example of Figure 2.4b depends on the pinned file line information to fully understand it. Fortunately, GitHub's user interface allows contributors to peek at the file's content around the pinned line and it solves the challenges that context-dependency imposes.

Finally, the example of Figure 2.5 shows a test review that is motivated by a change in the SUT. In this specific case, the reviewer is concerned about the referenced change being major since it breaks the existing behavior. Then, the reviewer suggests the

[3] demisto/content#2753


[redacted] requested changes on Sep 6, 2018

View changes

[redacted] left a comment

Owner

Finally, I would like to have your opinion if this should be a major change or not.

I am personally tending to keeping this as a minor change as it is just a feature addition.

lib/mongoose-field-encryption.js Outdated

```
-     if (encryptedFieldValue === false && plainTextValue) {
+     if (plainTextValue) {
```

[redacted] on Sep 6, 2018

Owner

It looks like the check for `encryptedFieldValue === false` is removed now. This is very necessary as the value might be double encrypted if it is passed to the update statement by mistake.

[redacted] on Sep 6, 2018

Owner

We should probably also add a test for this case :)

[redacted] on Sep 7, 2018

Contributor Author

You are right :) I'll try to cover it with a test somehow

Figure 2.5: Test review about API breaking change

implementation of a test for this case, so the test will detect when the change is applied.

2.4 Mutation Testing

Our proposed tool to assess test method quality relies on mutation testing. Therefore, we provide here an overview of this technique to facilitate understanding.

The mutation testing technique assesses test effectiveness in four major steps (illustrated in Figure 2.6). First, the project test suite is executed and the results are stored as the *expected output*. Then, a mutation testing engine (*e.g.*, PIT [16]) parses the code and applies mutation operators on code structures generating a set of mutants. The mutants are separately tested by the test suite and the results form the *obtained output*. Lastly, each *obtained output* is compared to the *expected output*. A mutant is *killed* when at least one of the test results differs between both sets, *i.e.*, when at least one of the test methods run on the mutants failed, meaning they properly detected the code mutations. Finally, a mutation score is computed: higher scores mean the test suite is better in catching real bugs [62].

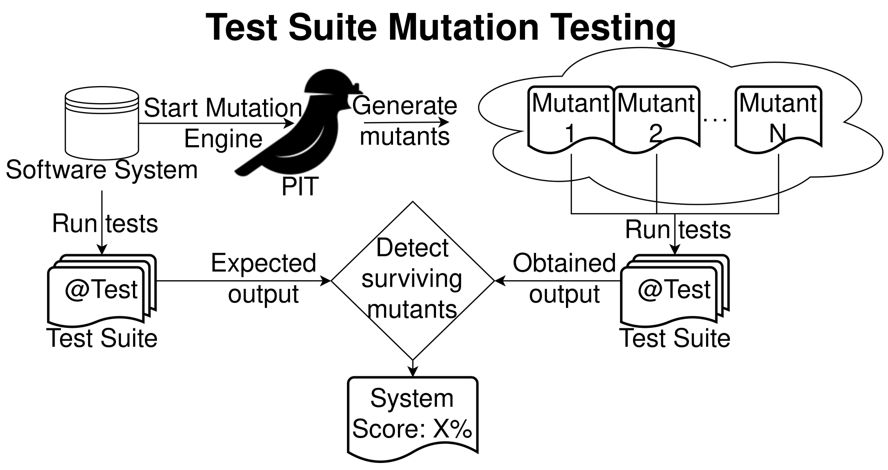
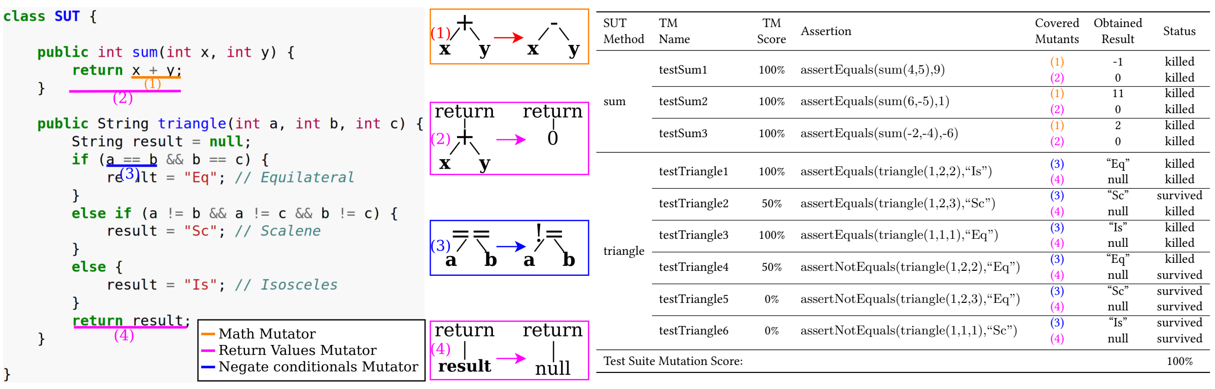


Figure 2.6: Traditional mutation testing approach

2.4.1 Mutation score computation

The mutation score is defined as the ratio of killed mutants and the number of generated mutants (which includes the *killed*, *survived*, and *uncovered* sets). A mutant is *killed* in three scenarios: *failure*, *error*, or *time-out*. A *failure* happens when the test fails, *i.e.,* an assertion detected the modification. An *error* occurs when an exception is raised. Lastly, a *time-out* happens when the test execution takes considerably longer, possibly leading to an infinite loop. *Survived* happens when the test passes, *i.e.,* no assertion detected the modification. Uncovered mutations cannot be killed, because no test run reached them, hence PIT skips their execution.

Figure 2.7 exemplifies the execution of mutation testing based on three mutation operators. The target system has a production class (`SUT`) with two methods, `sum()` and `triangle()`. It also has nine test methods: three cover `sum()` and six cover `triangle()`. For simplicity, we do not show the code of the test methods, but their asserts (column "Assertion"). The four generated mutants are annotated in the `SUT` source code and detailed in the boxes. For example, Mutation 1 replaces the "+" (sum) operator with "-" (subtraction). Also, for each test method, Figure 2.7 shows its related mutants, the obtained result, and the status of the mutant. At last, the mutation score for the target system is 100% because `testSum1()` kills mutants 1 and 2, while `testTriangle1()` kills mutants 3 and 4.



```
class SUT {

    public int sum(int x, int y) {
        return x + y;
    }

    public String triangle(int a, int b, int c) {
        String result = null;
        if (a == b && b == c) {
            result = "Eq"; // Equilateral
        }
        else if (a != b && a != c && b != c) {
            result = "Sc"; // Scalene
        }
        else {
            result = "Is"; // Isosceles
        }
        return result;
    }
}
```

(1)
(2)
(3)
(4)
— Math Mutator
— Return Values Mutator
— Negate conditionals Mutator
(1) + x y → - x y
return (2) + x y → return 0
(3) == a b → != a b
return (4) result → return null

SUT Method	TM Name	TM Score	Assertion	Covered Mutants	Obtained Result	Status
sum	testSum1	100%	assertEquals(sum(4,5),9)	(1)	-1	killed
				(2)	0	killed
	testSum2	100%	assertEquals(sum(6,-5),1)	(1)	11	killed
				(2)	0	killed
	testSum3	100%	assertEquals(sum(-2,-4),-6)	(1)	2	killed
				(2)	0	killed
triangle	testTriangle1	100%	assertEquals(triangle(1,2,2),“Is”)	(3)	“Eq”	killed
				(4)	null	killed
	testTriangle2	50%	assertEquals(triangle(1,2,3),“Sc”)	(3)	“Sc”	survived
				(4)	null	killed
	testTriangle3	100%	assertEquals(triangle(1,1,1),“Eq”)	(3)	“Is”	killed
				(4)	null	killed
	testTriangle4	50%	assertNotEquals(triangle(1,2,2),“Eq”)	(3)	“Eq”	killed
				(4)	null	survived
	testTriangle5	0%	assertNotEquals(triangle(1,2,3),“Eq”)	(3)	“Sc”	survived
				(4)	null	survived
	testTriangle6	0%	assertNotEquals(triangle(1,1,1),“Sc”)	(3)	“Is”	survived
				(4)	null	survived
Test Suite Mutation Score:						100%

Figure 2.7: Score computation example in mutation testing inspired by [62] (“TM”: Test Method)

2.4.2 Limitation of Test Suite Mutation Testing

Although test suite mutation testing is ideal for gathering the overall test quality in a system, it has three limitations: (1) the overall system mutation score overshadows the quality of individual test methods; (2) the quality of a contribution (*e.g.*, a pull request with code and tests) can be unnoticed in a large system because its score may be unaffected by small code changes (due to existing tests outnumbering the contributed tests); (3) existing test methods may kill mutants within a contribution and hinder assessing the quality of the contributed tests. For instance, in the previous example, `testTriangle5()` and `testTriangle6()` kill no mutant, suggesting they are the most fragile contributed test methods. However, the system score is unaffected because their mutants are killed by other tests. Thus, the 100% mutation score neglects the quality difference between the test methods, from the mutation analysis perspective.

2.5 Related Work

2.5.1 Code Review

Spadini et al. [58] assessed the particularities of code reviews targeting automated test code rather than production source code. The authors found no association between future defects and the type of code (*i.e.*, production or test). Furthermore, they found reviews are more likely to discuss production code than test code when both are present.

However, test-only code reviews have more comments and reviewers. Also, the most discussed topics during reviews are path coverage (especially corner cases), testing practices, complexity, maintainability, and readability of tests. Also, the authors classified the comments in five out of eight categories from prior studies [48] and further extracted finer-grained information for each. Despite the three missing categories, the others (*i.e., code improvements*, *understanding*, *social communication*, *defect finding*, and *knowledge transfer*) were similarly frequent. In this study, we find requests for *change in existing tests* whose sub-categories partially intersects those outcomes, *e.g., refactor test* and *improve test* relate to *code improvements*, and *fix test* relates to *defect finding*.

Spadini et al. [58] also interviewed practitioners to study whether tests should be reviewed first and the results were further expanded in another study [63]. Four arguments about reviewing tests before production code were extracted from the interviewees' responses from the first study: two in favor (*i.e.,* "*understand the API first*" and "*check SUT does only what is tested for*") and two against (*i.e.,* "*tests are usually very bad*" and "*prior understanding of what should be tested is faster*") [58]. The second study involved an experiment to assess the influences of the review order on the review effectiveness. The results demonstrated no statistical difference between test-first and production-first review effectiveness. However, they found that test-first reviews led to more bugs found in test code and fewer maintainability issues in production code. The authors conclude some practitioners see tests as less important than production, thus they have limited time to spend reviewing tests [58, 63]. Also, tests are harder to review than production code because of their lower quality. Nevertheless, developers perceive Test-Driven Code Review "*helps teams being more testing-oriented, hence improving the overall test quality*" [63].

The definition of Modern Code Review is threefold [48]: informal, tool-based, and regularly practiced. Bacchelli and Bird inspected Modern Code Review comments of sixteen product teams at Microsoft and surveyed 165 managers and 873 programmers [48]. They found practitioners' motivations and expectations for reviewing code did not match its actual outcomes. Although *finding defects* was the leading motivation to review code, it was only the fourth most common comment subject (behind *social communication*, *understanding*, and *code improvements*). Furthermore, the authors present challenges to "*understand the reason of a change*" as the main reason for that difference.

Wen et al. [64] studied the changes of code review over time in Dell EMC and OpenStack, which are respectively a big proprietary organization and a solid open-source community. The authors created a model using LDA to identify discussion topics, grouped the topics into seven high-level groups, and analyzed the prevalence of each group in terms of both the overall community maturity and the individual experience of developers. They found both projects present similar trends over time and reviewers tend to stabilize in some topics according to their communities' needs.

Sadowski et al. [37] presented a case study of the Modern Code Review practices adopted at Google. They qualitatively and quantitatively analyzed data from several distinct sources (i.e. Interviews, surveys, and logging history) and identified five key themes during code reviews at Google: Education, maintaining norms, gatekeeping, accident prevention, and tracking history. Also, the authors found review themes depend on the relationship between the author and the reviewer. The paper expands on prior knowledge regarding how Modern Code Review speeds up feedback and requires fewer reviewers, as evidenced by Google taking significantly less time (4 hours vs. 14.7 hours [65]) and only involving half the amount of reviewers on average (1 vs. 2s [65]).

McIntosh et al. [66] analyzed three open-source projects (ITK, VTK, Qt) that adopt Gerrit as a code review tool and have many reviewed patches. The analysis suggested that, for some open-source projects, low review coverage, no discussion, and lacking subject matter experts contribute to a higher defect-proneness. Nevertheless, the authors did not find a consistent relationship between post-release defects and discussion length throughout the studied projects. In this study, we assess code review discussions' content and measured how information richness contributes to a pull request acceptance rate.

Studies suggest Pull-Based Development is the next trend in the software industry due to the increase in popularity of platforms supporting that development model [45, 57]. Gousios et al. searched pull requests on the GHTorrent dataset and assessed both the integrator's and contributor's perspectives. They observed most integrators (80%) use pull requests for code review, use the web interface to integrate patches without losing commit metadata, and inspect *code quality*, *code style*, *project fit*, *testing*, and *documentation* (in this order). In this study, we further assess one of these aspects, *testing*, which is also the main mechanism applied by contributors to ensure the quality of their own contribution [57]. Whereas the integrators examine quality in terms of style and design conformance, code quality, and test coverage. While CI's adoption is substantial (75%), dedicated software quality tools are rarely adopted. The authors also found integrators and contributors share social challenges, *e.g.,* responsiveness and reaching consensus.

Ebert et al. [67] analyzed 499 questions in Android's code reviews from Gerrit and classified them into five communicative intentions types, of which *request for action* was the prevalent, followed by *ask for clarification* of the reviewer's understanding. Communicative intention and lack of rationale, context, documentation, familiarity (with the code), and tests are reasons for confusion in code review and their outcomes are delays, blind approval, confidence loss, and contribution abandonment [61]. In this study, we assess unsolved test requests, which highlight other aspects of contribution abandonment by its author. Nonetheless, Alami et al. found open-source communities embrace rejection as part of the contribution review process [7]. Lastly, beyond code review, an experiment showed that task granularity assists novice developers in building complete solutions using TDD [68]. Our results support that finding is expandable to requests within a code review discussion.

2.5.2 Mutation Testing

We recall that in our second study we propose a tool that relies on mutation testing to measure the quality of the finest grained unit in testing, the test method. Furthermore, measuring test suite effectiveness through mutation testing is a largely studied topic with well-defined benefits and constraints [69]. Addressing those points, Jia and Harman [69] summarize 390 studies in a public repository. On the other hand, Grano et al. [70] found that researchers and practitioners perceive existing metrics assist in detecting low-quality test suites, but do not guarantee the high quality of a test suite [70].. Catolino et al. [71] find assertion density correlation with developer's experience and class-related factors.

Test smells are associated with several factors in software development, for example, code smells [72], change-proneness, defect-proneness [73], and post-release defects [74]. Despite the richness of the test smell research topic, practitioners do not perceive test smells as actual problems [72, 74], 90% of test smells are never fixed, and fixing takes, on average, 100 days [72].

Hilton et al. assess the impact of finer granularity reports on some test coverage limitations. They describe how non-code changes impact test coverage and how finer granularity reports lead to better understanding of the quality of a specific change [75]. Despite being considerably cheaper than mutation testing, test coverage still has challenges when adopted in large-scale projects [76]. Challenges aggravated by the monorepo settings at Facebook, where test prioritization usage reduced infrastructure overhead [76].

2.6 Final Remarks

In this chapter, we presented key concepts related to this dissertation. Specifically, we detailed the concepts of code review, Modern Code Review, test review, and mutation testing. Finally, we present work related to the major themes of this dissertation, involving test reviews and mutation testing.

Chapter 3

How Developers Review Tests on GitHub

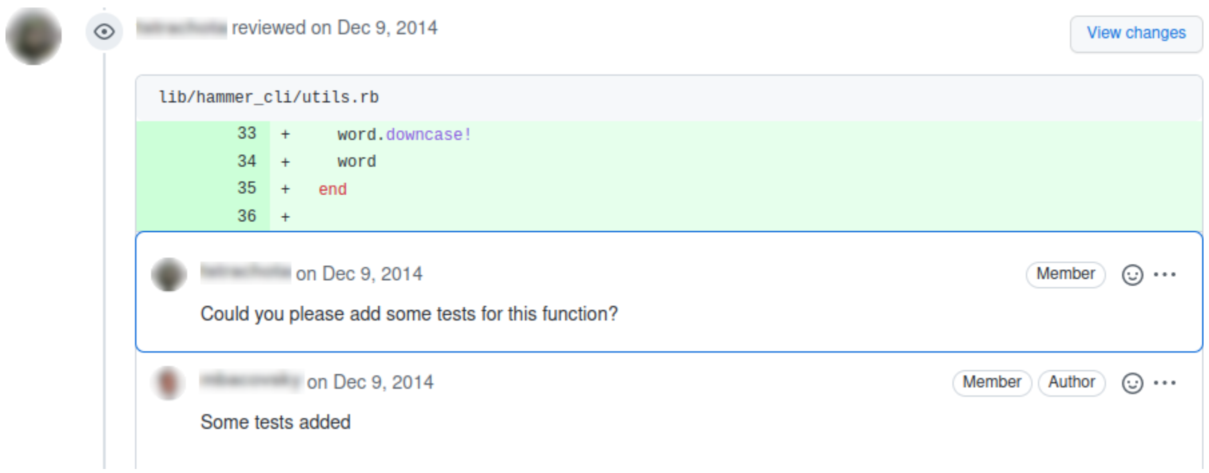
In this chapter, we present the main study of this dissertation, which evaluates how developers review tests on GitHub. This chapter is organized as follows: Section 3.1 details the study design, Section 3.2 reveals the results, Section 3.3 features discussions and implications, and Section 3.4 discusses this study's threats to validity.

3.1 Study Design

3.1.1 Collecting Test Reviews

We aim to study relevant software projects and test reviews on GitHub. For this purpose, we rely on the GHTorrent [15] dataset to collect projects and their respective test reviews. First, we need to search for pull requests with evidence of test reviews. Hence, we aim to find pull requests that include sentences like "*could you please add tests*" and their corresponding discussions. For example, Figure 3.1 shows a test review for the contribution from pull request #155 of project theforeman/hammer-cli on GitHub.[1] The reviewer *tstrachota* pinned line 33 of file utils.rb and asked for test addition to the contribution acceptance. The contributor *mbacovsky* replied positively, stating he implemented a test covering the function located at that specific changed line of code.

[1]github.com/theforeman/hammer-cli/pull/155



[redacted] reviewed on Dec 9, 2014

lib/hammer_cli/utils.rb

```
+     word.downcase!
+     word
+   end
+
```

[redacted] on Dec 9, 2014

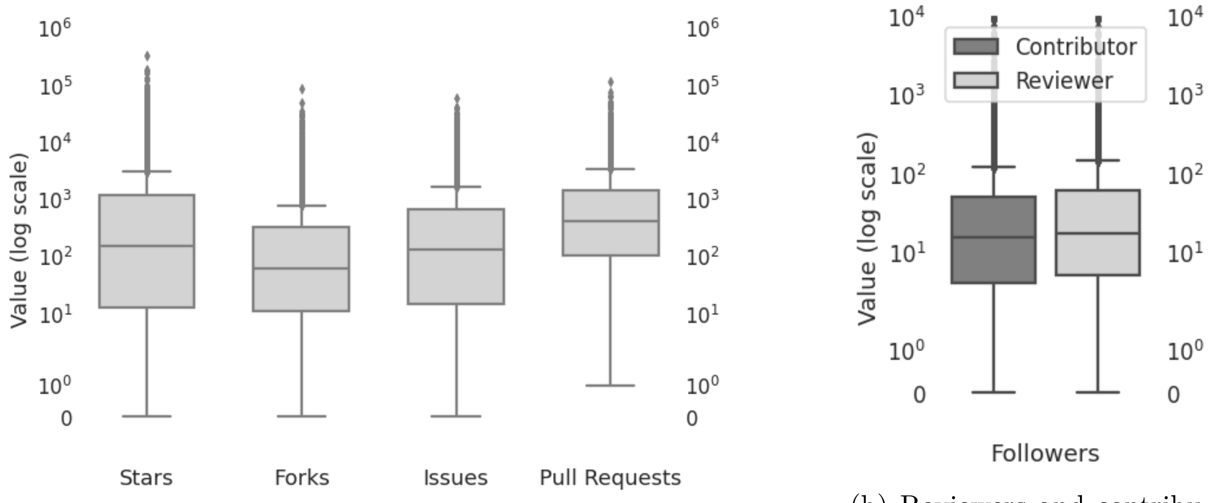
Could you please add some tests for this function?

[redacted] on Dec 9, 2014

Some tests added

Figure 3.1: Request for tests covering a specific function

To find test reviews, we query the GHTorrent dataset for projects with pull requests with the following regular expression: $can|could|please|should|would|consider$, followed by a word related to an addition $add|adding$, and ending with the word *test*. We find 11,836 candidate test reviews in 5,421 projects. Figure 3.2a presents the distribution of the number of stars, forks, issues, and pull requests for those projects. On the median, the selected projects have 151 stars, 61 forks, 132 issues, and 417 pull requests. Figure 3.2b presents the distribution of the number of followers for both test contributors and reviewers. On the median, the contributors have 15 followers and the reviewers have 17 followers



(a) Projects

(b) Reviewers and contributors

Figure 3.2: Overview of projects', contributors', and reviewers' social metrics

3.1.2 Manual Evaluation of the Test Reviews

In the previous section, we found 11,836 candidate test reviews. Next, we describe our process to assess and evaluate those test reviews.

Figure 3.3 presents an overview of our filtering steps. First, we start with 11,836 candidate test reviews, which are explored in RQ1. Second, we apply random sampling with a 95% confidence level and 5% confidence interval, resulting in a sample of 373 test reviews with 324 (86.9%) true positives, which we further explore in RQ2 and RQ3. Finally, we filter out the unsolved requests, assessing the remaining 252 test reviews in RQ4. In the following subsections, we detail each step.

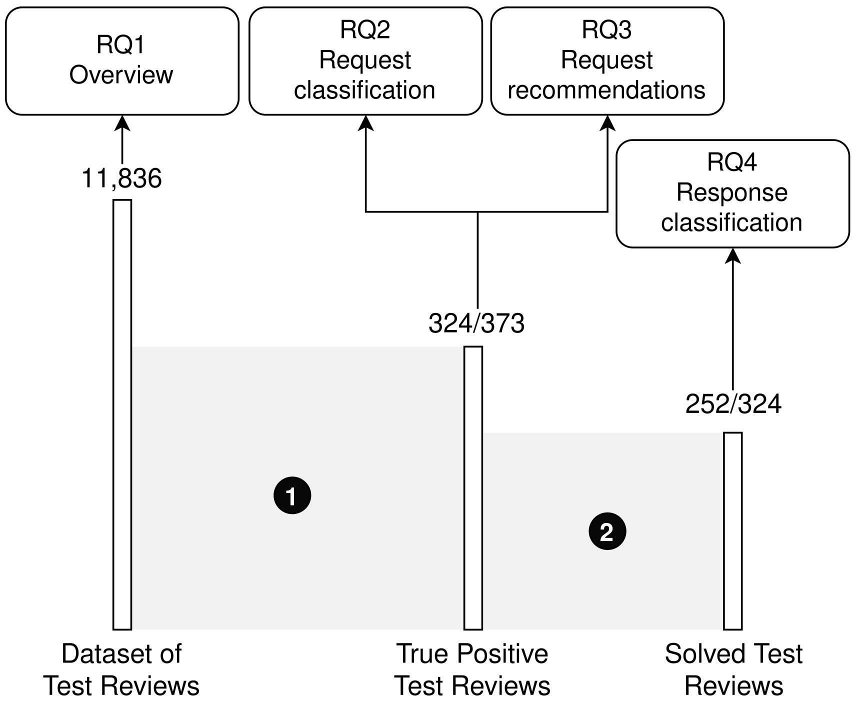

RQ1 Overview
RQ2 Request classification
RQ3 Request recommendations
RQ4 Response classification
11,836
324/373
252/324
1
2
Dataset of Test Reviews
True Positive Test Reviews
Solved Test Reviews

Figure 3.3: Overview of the study design

Query Validation

To create and validate our query, we set a fair precision goal of 85%, meaning that at least 4/5 out of the examined test reviews should be valid ones. Figure 3.4 summarizes this process, which includes five steps (from A to E). Next, we detail each step.

(A) Given the current version of the search query, we execute it in the Google Cloud Big Query in the GHTorrent latest public dataset.[2] (B) We apply random sampling with a 95% confidence level and 5% confidence interval. (C) We carefully manually evaluate each sample set and compute its precision in detecting true test reviews. (D) If the obtained

[2]https://twitter.com/ghtorrent/status/1222529377629605889

precision does not satisfy our goal (*i.e.,* 85%), we search for new patterns to improve our search query. (E) Finally, we improve our query by incorporating the patterns in it. This process was repeated five times until we end up with a reasonable precision of 86.9%.

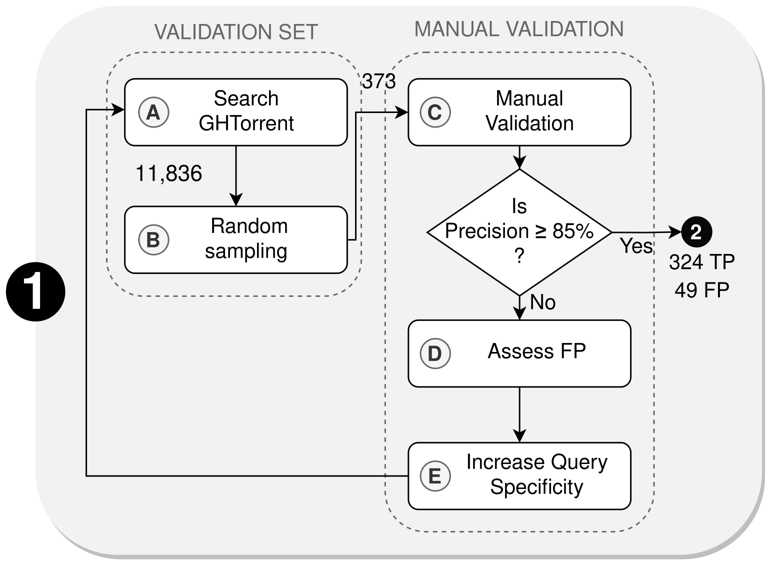
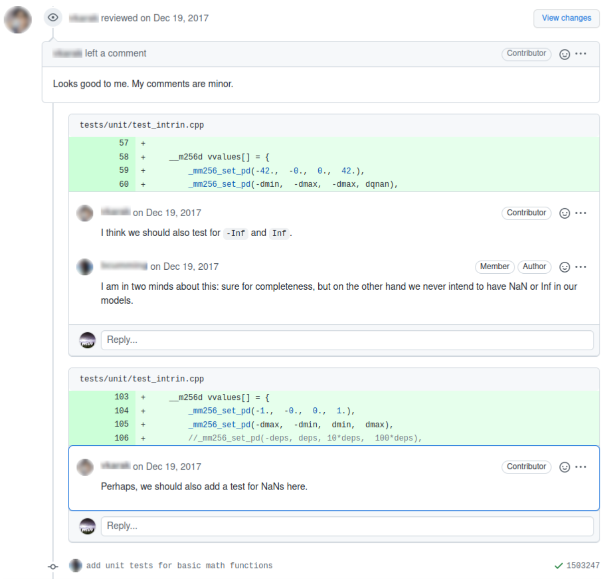


Figure 3.4: Query Validation Steps

Request/Response Classification

We represent the classification step in Figure 3.6. First, we collect 324 out of 373 true positive test reviews from the validation dataset. We classify the test reviews into two categories regarding their request: *add test* or *change existing test*. Figure 3.5a shows an example of the category *add test*, in which there is a request to add three specific cases. Figure 3.5b presents an example of *change existing test*, in which there is a request to move the unit test to another file.


[redacted] reviewed on Dec 19, 2017

[redacted] left a comment

Looks good to me. My comments are minor.

tests/unit/test_intrin.cpp

```
+
+     __m256d vvalues[] = {
+         _mm256_set_pd(-42.,  -0.,  0.,  42.),
+         _mm256_set_pd(-dmin,  -dmax,  -dmax, dqnan),
```

[redacted] on Dec 19, 2017

I think we should also test for `-Inf` and `Inf`.

[redacted] on Dec 19, 2017

I am in two minds about this: sure for completeness, but on the other hand we never intend to have NaN or Inf in our models.

tests/unit/test_intrin.cpp

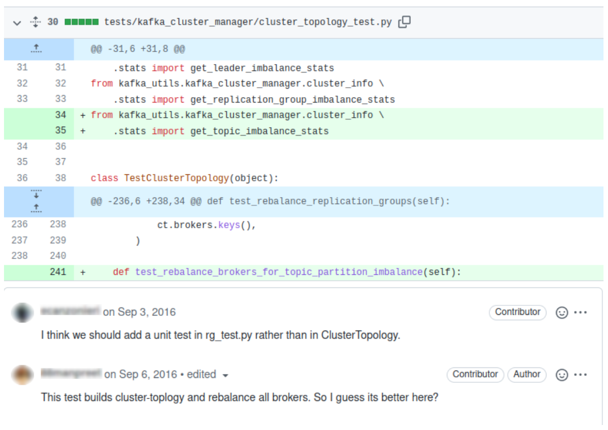
```
+     __m256d vvalues[] = {
+         _mm256_set_pd(-1.,  -0.,  0.,  1.),
+         _mm256_set_pd(-dmax,  -dmin,  dmin,  dmax),
+         //_mm256_set_pd(-deps, deps, 10*deps,  100*deps),
```

[redacted] on Dec 19, 2017

Perhaps, we should also add a test for NaNs here.

add unit tests for basic math functions ✓ 1503247

(a) Add Test (Edge Cases: -Inf, Inf, and NaN)


30 🟩🟩🟩🟩🟩 tests/kafka_cluster_manager/cluster_topology_test.py

```
  @@ -31,6 +31,8 @@
      .stats import get_leader_imbalance_stats
  from kafka_utils.kafka_cluster_manager.cluster_info \
      .stats import get_replication_group_imbalance_stats
+ from kafka_utils.kafka_cluster_manager.cluster_info \
+     .stats import get_topic_imbalance_stats


  class TestClusterTopology(object):
  @@ -236,6 +238,34 @@ def test_rebalance_replication_groups(self):
              ct.brokers.keys(),
          )

+     def test_rebalance_brokers_for_topic_partition_imbalance(self):
```

[redacted] on Sep 3, 2016

I think we should add a unit test in rg_test.py rather than in ClusterTopology.

[redacted] on Sep 6, 2016 • edited

This test builds cluster-toplogy and rebalance all brokers. So I guess its better here?

(b) Change Existing Test (Move Method Refactoring)

Figure 3.5: Examples of test review

To better understand the feedback provided in the test reviews, we manually assess each test request looking for testing recommendations provided by the reviewers. For example, a reviewer may suggest adding a mock to isolate the test or a cache to improve its performance. In Figure 3.5a, for instance, the test review is suggesting to cover an edge case.

Finally, we verify whether the test review was solved or unsolved. Solved means that the requested test was implemented and unsolved means that that the test review was rejected or ignored. For this purpose, we manually inspect the test reviewers to find the test code and the corresponding system-under-test (SUT). We find that 252 out of 324 (77.$\overline{7}$%) test reviews were solved, while 72 out of 324 were unsolved (22.$\overline{2}$%).

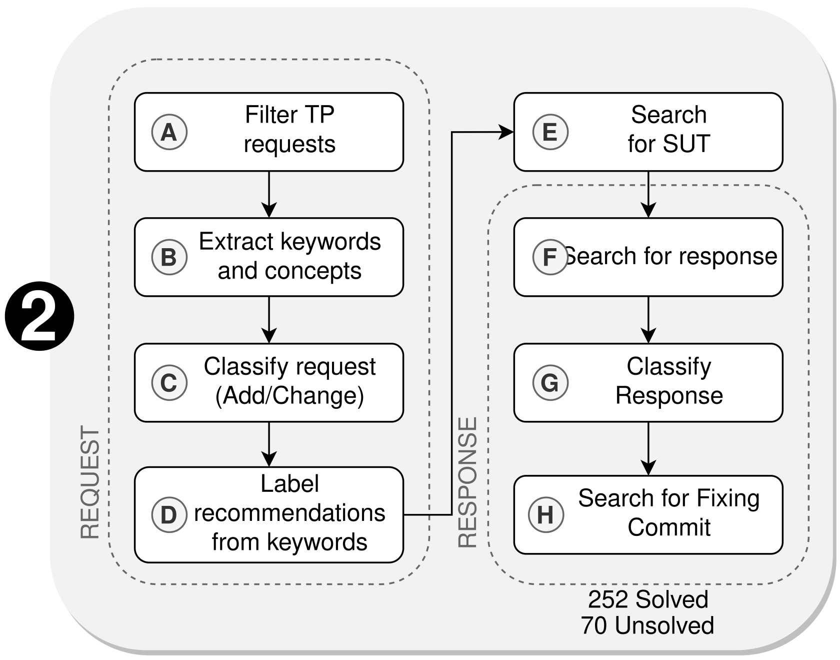

2
REQUEST
A Filter TP requests
B Extract keywords and concepts
C Classify request (Add/Change)
D Label recommendations from keywords
RESPONSE
E Search for SUT
F Search for response
G Classify Response
H Search for Fixing Commit
252 Solved
70 Unsolved

Figure 3.6: Discussion's content analysis and classification steps

3.1.3 Research Questions

RQ1: Test Review Extension

To better understand the extension of test reviews on GitHub, in this first research question, we explore the 11,836 test reviews described in Section 3.1.1. Specifically, we assess their frequency per project, the prevalent programming languages, and the project popularity. **Rationale:** We aim to discover to what extent test reviews are performed in the GitHub ecosystem.

RQ2: Test Request Types

In this second research question, we assess the reviewer's request in the test review. Specifically, we explore what are the types of test requests in the 324 test reviews described in Section 3.1.2. In this process, we find two types of test requests: *add test* and *change existing test*. *Add test* represents cases in which the reviewers ask for the implementation of non-existent tests. *Change existing test* includes cases in which existing tests should either be refactored, fixed, or improved. **Rationale:** We aim to find what are the common limitations of the tests present in the process of test review. For example, are there more requests to add or change tests? What types of changes are more common?

RQ3: Test Request Recommendations

In this research question, we continue to assess the reviewer's request in the test review. Here, we focus on exploring what specifically the reviewers suggest to the contributors so that they can improve their tests. For simplicity, we call those suggestions *test recommendations*. Here, we also rely on the 324 test reviews described in Section 3.1.2. To analyze the test requests qualitatively, we apply *thematic analysis* [77] on each test request. The analysis seeks to identify and record themes in textual documents, using the following steps: (1) initial reading of the test requests, (2) generating a first code for each test request, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes. The first three steps were carried out by the first author of this dissertation, while steps 4 and 5 were developed by the consensus of the two authors through several meetings and discussions. **Rationale:** We aim to find what reviewers commonly suggest to improve the tests in test reviews. This information is important to elaborate on the usual limitations of the tests in test reviews as well as the usual recommendations of reviewers.

RQ4: Test Responses

Finally, in our last research question, we study the contributors' response to a test request. Specifically, we assess whether the request is solved or unsolved. We rely on the 252 solved and 70 unsolved described in Section 3.1.2. **Rationale:** We aim to explore what requests are easier and harder to solve.

3.2 Results

3.2.1 RQ1: How frequently do reviewers ask contributors for tests?

We find 11,836 *test reviews* in 5,421 GitHub repositories. Figure 3.7 presents the number of test reviews per programming language. Here, we filter out programming languages with less than 300 test reviews. The most common language is Python, which contains 2,187 test reviews. Next, we have Java and JavaScript with 1,674 and 1,644 cases, respectively. The other six programming languages are Go (1,304 test reviews), TypeScript (737), C++ (717), C# (624), Ruby (601), and PHP (356). Notice that ranking does not reflect the popularity of such languages on GitHub. For instance, JavaScript is the most popular programming language on GitHub, while we find JavaScript is the third language most discussed in *test reviews*. Therefore, there may be some social features impacting how likely a community address test code during code reviews.

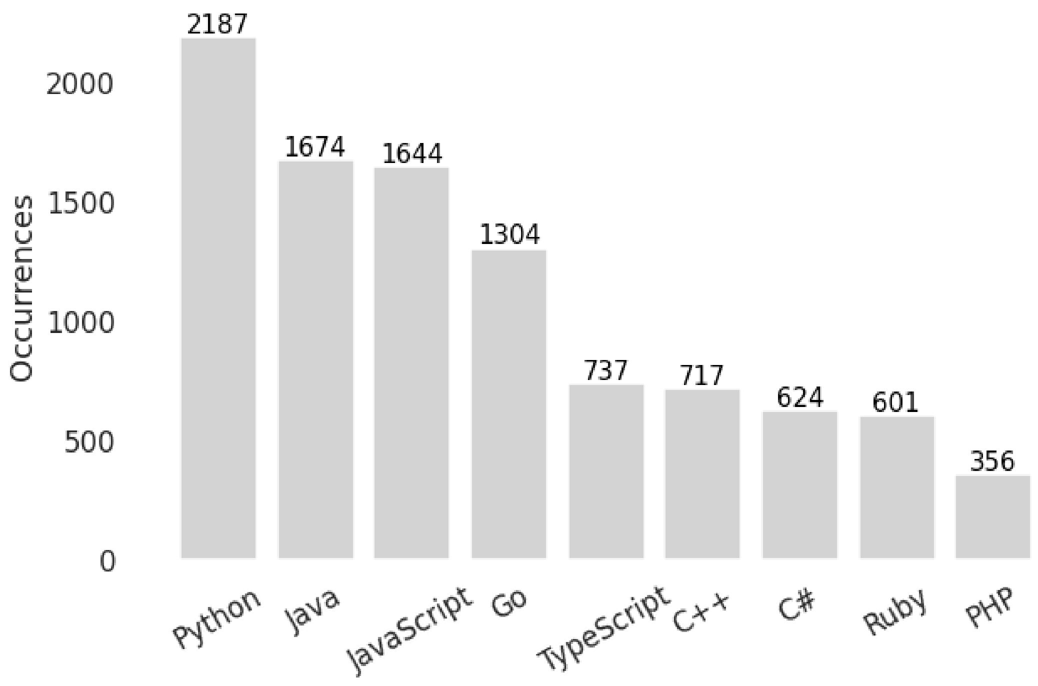


Figure 3.7: Distribution of test reviews per programming language

Table 3.1 summarizes the top-10 projects with the majority of the test reviews. Those are relevant projects, with thousands of stars and pull requests (PRs). *Pandas* is the top-1, with 132 test reviews (1.1%). Next, we have *Roslyn* and *Kubernetes*, with 120 and 114 respectively. The top-10 comprises several important projects, such as *ElasticSearch* (111), *mypy* (73), and *Presto* (57), and represents 7.2% of the entire dataset. In addition, we add the total of *test reviews* for the top-100 (2,784 or 23.5%), top-1000 (6,812 or 57.6%), and for the whole dataset (11,836).

Summary: Test reviews are indeed performed on popular programming languages and projects. We find 11,836 test reviews in 5,421 open-source repositories.

Table 3.1: Top repositories

Repository	Language	Stars	PRs	Test reviews
pandas-dev/pandas	Python	31K	23K	132
dotnet/roslyn	C#	15K	29K	120
kubernetes/kubernetes	Go	82K	66K	114
elastic/elasticsearch	Java	57K	51K	111
dotnet/corefx	C#	18K	25K	108
python/mypy	Python	11K	5K	73
prestodb/presto	Java	12K	12K	57
ampproject/amphtml	JS	15K	23K	52
envoyproxy/envoy	C++	18K	12K	47
edx/edx-platform	Python	6K	30K	43
Top-10	-	-	-	857
Top-100	-	-	-	2,784
Top-1000	-	-	-	6,812
Total	-	-	-	11,836

3.2.2 RQ2: What do reviewers ask contributors for?

In this research question, we start to analyze the content of the test reviews. In our manual analysis, we find two types of test reviews: (1) with request to *add tests* and (2) with request to *change existing tests*. Table 3.2 summarizes the frequency of each test request. We detect nine categories of test requests: six related to *add test* (272, 83.7%) and three related to *change existing test* (53, 16.3%).

Table 3.2: Test request classes, their categories, and frequencies

Category Classes	Frequency
Add Test	**272 (83.7%)**
Cover a case	140
Unspecified SUT	47
Cover a method	43
Cover a class	17
Cover a statement	13
Cover a branch	12
Change Existing Test	**53 (16.3%)**
Refactor test	20
Improve test	20
Fix test	13

The six *add test* categories are based on the SUT granularity, as follows:

- *Statement* is the finest grained and is a single line in the source code.
- *Branch* is coarser than a *statement* and is a code block that belongs to a loop or conditional.
- *Method* is even coarser, *i.e.,* it may contain one or more branches and statements.
- *Class* is the coarsest grained SUT.
- *Case* is a special classification for requests specifying the test case's input or setting.
- *Unspecified* contains all *added tests* without an explicit SUT.

Likewise, we study three *change existing test* categories, as follows:

- *Refactor test* requests present test changes without affecting its observable behavior.
- *Improve test* requests are opinionated and ask for changes on correct but poorly designed test code.
- *Fix test* requests objectively target well-known bad practices revision or minor mistakes correction.

Next, we explore the number of recommendations provided by the reviewers on each request. We recall that the recommendations are further explored in RQ3.

In Figure 3.8, we present the distribution of recommendations for *add test*. The number of recommendations ranges from 0 to 1, on the first quartile, and from 1 to 2, on the third quartile. This shows that recommendations are more frequent on *case*, *unspecified*, and *statement* than on *method*, *class*, and *branch*. Moreover, on the median, all *add test categories* have one recommendation, except *branch* which has 0.5.

Lastly, Figure 3.9 presents the distribution of recommendations for *change existing test*. Overall, this distribution is more homogeneous than the *add test*. For instance, all three categories have approximately two as the third quartile, one as the median, and one as the first quartile.

Summary: Test reviews contains requests to add tests and change existing tests. We find that both contains recommendations for guide developers. Requests to add test frequently present distinct level of SUT granularity, from branch to class.

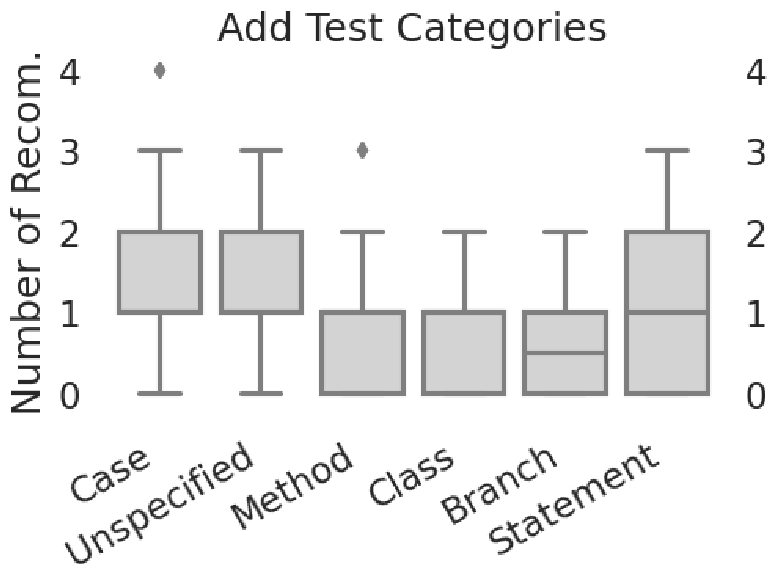


Figure 3.8: Distribution of recommendations in *add tests*

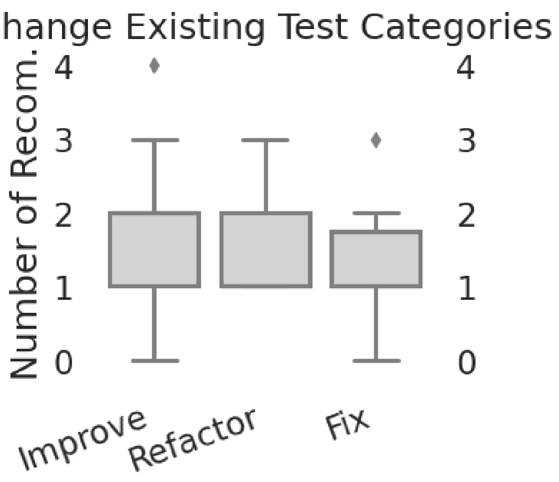


Figure 3.9: Distribution of recommendations in *change existing tests*

3.2.3 RQ3: Which recommendations are frequently found in test requests?

In this research question, we manually analyze the reviewers' request to assess their recommendations. As presented in Table 3.3, we find seven broad categories of recommendations: *test scope*, *tool support*, *test scenarios*, *goal/purpose*, *refactoring*, *bad practices*, *fixtures*, and *miscellaneous*. Those broad categories are composed of 36 specific recommendations that the contributors should focus to improve their tests. Next, we describe each broad category with real examples.

Table 3.3: Recommendations within a test request, their categories, and frequencies

Category Recommendations	Description	#
Test Scope		**81**
Unit	Tests a single unit	42
End-to-end	The test crosses all system's layers	31
Integration	Tests the integration between components	8
Tool Support		**39**
Parameterized test	Test reused for multiple cases	16
Test double	Relies on mocks/stubs/spies/fakes	12
Async	Targets asynchronous design challenges	4
Threading	Concerns multi-threading/thread-safety/delays	4
Cache	Tests cache structures' behaviour	3
Test Scenarios		**118**
Specific snippet	Specify test code fragment	40
Edge case	Uncommon case requiring special handling	26
Expected exception	Expects SUT to throw an exception	25
Negative case	Verify unexpected or false outputs	12
Positive case	Relies on positive assertions	8
Corner case	Tests edges of input's equivalence partitions	4
Compilation check	Tests cases that only exercises static checks	3
Goal/Purpose		**13**
Reproduce issue	Guarantees the solved issue does not regress	5
Prevent regression	Acknowledge regression concerns	4
Prevent inconsistency	Detects broken contracts/naming conventions	4
Refactoring		**26**
Increase testability	Simplify SUT's for testing	6
Rename test	Replace test case's name or description	6
Improve assert message	Add/polish assertion's explanation parameter	4
Move test	Reposition a test case within a test suite	4
Extract member	Extract method or field for reuse purpose	3
Merge test	Brings code from multiple test cases together	3
Bad Practices		**15**
Unneeded test	Internal/deprecated/unneeded API testing	7
Generic test	Tests should be more specific and stricter	5
Empty test	Delete or populate empty tests	3
Fixture		**27**
Setup	Informs required pre-conditions	20
External resource	Involves external files or databases	7
Misc.		**51**
Filepath	Informs a path for the test case	20
Type checking	Type support in dynamically-typed source code	9
Archetype test	Example test provided for inspiration	8
Event test	Triggers or listens to events	5
Dependency	Asks for dependency or environment changes	4
Modifier	Involves annotation or access modifier changes	3
Increase readability	Facilitate comprehension of the test code	2

Test Scope (25%)

Test scope represents the three layers in the test pyramid: *unit*, *integration*, and *end-to-end* tests). The prevailing recommendation is related to *unit test*, with 42 occurrences.

For example, a reviewer requested a *unit test* in BaGet's repository at pull request #162 states: "*Also, please add a unit test in* `FileStorageServiceTests.cs`" (BaGet#162). The coarsest grained *test scope*, *end-to-end*, comes afterward with 36 occurrences. Due to different project settings and software architecture, the concept of end-to-end may vary from project to project. Indeed, in Mymove's repository, there is a test review (mymove#199) whose reviewer reinforced the importance of keeping the *test scope* consistent: "*This test no longer tests the award queue 'end to end...' We should either rename it or add that functionality to this test.*" *End-to-end* tests may also be called *Karma tests* by the JavaScript (and TypeScript) community, as seen at oppia#5724: "*Probably should add karma tests to check this.*" Surprisingly, *integration test* is the least requested *test scope* by a wide margin, and its frequency adds up to eight.

Tool Support (12%)

Tool support represents suggestions of popular technologies to be used in the test implementation. Test frameworks provide many features to support test development with enhanced isolation and reusability. The most commented one is the *parameterized test* with 16 occurrences, like the one we find at runtimes-common#210: "*Perhaps you can add an additional parameter to your test cases,* `shoulderror` *and you can try to test some of the error cases for your code as well.*". The next one is *test double*, *e.g.*, mocks, stubs, spies, and fakes, which appears 12 times. An example of a test request with *test double* recommendation is: "*can we add a test with spies, for each of these branches, to ensure it's called both once, and in the proper order?*" (found at gutenberg#6231). Then, two categories, *async* and *threading*, have the same frequency: four times, each. Their frequency is not the only thing they have in common, both techniques are lightweight alternatives to multi-processing and offer challenges to be faced during testing. For instance, at react-component-variations#15 the reviewer's concern refers to improper use of async in production, while at synapsePythonClient#661 the reviewer emphasizes possible differences in infrastructure between test and production environments that may lead to challenges with parallelism. Lastly, *cache* is the least frequently mentioned *tool support* with only three occurrences.

Test Scenarios (36.4%)

The second most frequent category is *test scenarios* and it is the category with most recommendations (seven). Here, reviewers ask contributors to focus on specific scenarios when creating their tests. The most frequent, *specific snippet*, is also the most specific scenario as it involves test requests containing code snippet like "`expect(page.current_path).to eq(organization.user_admin_path)`", which is found at caseflow#9288. The next one is *edge case*, which is normally a specific scenario the developers did not consider yet.

It can be either specified like at stitch-android-sdk#90 or be a general demand like at Python#794 in which the reviewer said: "*Also, you may consider to add a few more test cases, like edge cases.*" And the latter, *expected exception*, tests throw statements to evaluate whether production code handles expected failures. It is commonly addressed by test developers in open-source projects [78, 79] and we find two examples of test requests with this recommendation in the *manageiq-api* project: #313 and #245. Among the least frequent *test scenarios*, there are two representing opposite assertion types *negative case*, with 12 mentions, and *positive case*, with 8 mentions, that were named after studies in the literature [80, 81]; *corner case*, that occurs four times and is actually a common strategy to reduce the number of tests; and finally, the weakest and most infrequent (with three occurrences) is the *compilation check*.

Goal/Purpose (4%)

We also find recommendations in which reviewers make it clear their goal or purpose. The primary externalized motivation for a test request is *reproduce issue*. Although *reproduce issue* and *prevent regression* are common testing *goals* [82, 83], reviewers seldom explicitly mention them (respectively, 5 and 4). "*Great, can you add some test for the bug you fixed? thanks.*" ng-zorro-antd#2136 and "*LGTM, can you add this test case I wrote to reproduce the issue?*" (interop#133) are examples of requests to *reproduce issue*. Furthermore, reviewers explicitly ask tests to *prevent inconsistency* as often as they do so to *prevent regression* (four times), and sometimes both *goals* are identified: "*Since this is a primary key for joining across data sets, you could add a consistency test. Should be easy, and would catch breakage of this in future.*" (gocd#468).

Refactoring (8%)

Refactoring comprises test-related refactoring. One of the most frequently requested refactorings targets the SUT instead of the test code itself, occurs six times in our validation set, and aims to *increase the SUT's testability*. For example, at fusor#1223 the reviewer asked to expose some methods by setting them up as static (by prefixing `self.` to their definition's name) and extracting an utility class with them: "*I would've put all of this logic in a utility class* `fusor/server/app/lib/utils/fusor` *Seems like it could be generally useful in other places if need be. And it would be easy to add a unit test around it, and makes the deployment_controller smaller.*" Another refactoring as frequent as *increase testability* is *rename test*, that is a testing variation of Fowler's *rename method* refactoring. The same applies to *move test*, *extract member*, and *merge test*, which happen 4, 3, and 3 times respectively. Lastly, we identify *improve assert message* as a test-specific refactoring targeting readability that occurs four times in our validation set.

Bad Practices (4.6%)

Bad Practices include recommendations advising against bad test smells, such as testing internal/deprecated/unneeded API and badly written tests. Test reviewers revealed concerns about *unneeded tests*, such as testing internal, deprecated, or trivial API. For instance, at refined-github#1255, when the reviewer asked about the motivation for such fine-grained methods, the contributor answered: "Someone might need to use it," and the reviewer rebutted: "They'll add it if they'll need it. Currently two of these new is aren't tested. It can just be renamed (or you can add the tests)." Moreover, *generic tests* represent one-third of the mentioned *bad practices* in test requests and 1.5% of all the recommendations occurrences. An example of *generic test* was committed at odoo#25047: "*Also, I would add more assert in this test, because it may easily pass successfully even though there's nothing that has been tested (...). That way, you have more confidence that the data are correctly configured.*" Besides, *empty tests* is the least discussed *bad practice* (20% of the *bad practices* and 0.92% of all recommendations).

Fixture (8.4%)

Fixture involves solely fixture located warnings. The *setup fixture* recommendation is considerably more frequent than *external resource*. Among its 20 occurrences, we emphasize a case (pertax-frontend#220) in which the reviewer suggested extracting a common local variable as a field initialization at *setup*. On the other hand, *external resource* happens seven times (*e.g.,* centraldogma-go#9) and it is associated with the file system usage (*e.g.,* file generation, file validation, and database usage).

Misc. (15.7%)

The frequency of the remaining recommendations varies a lot (from two to twenty), whereas there are three leading ones: Twenty test requests include a *filepath*, *e.g.,* support-frontend#366 in which the reviewer asked "*Can you please add test for this case in* `bundlesLandingReducersTest.js`*?*" Nine test requests, such as pandas#23262, explicitly verify for type support in dynamically typed languages. Other eight mention examples of tests the contributor should use for inspiration, we call them *archetype tests*. One example, whose reviewer asked for a test based on `IndexShardTestCase`, is found at elasticsearch#24858. At last, four less frequent recommendations are: *event test*, *dependency*, *modifier*, and *increase readability*.

Summary: We find seven broad categories of recommendations: *test scope*, *tool support*, *test scenarios*, *goal/purpose*, *refactoring*, *bad practices*, *fixtures*, and *miscellaneous*. Those broad categories are composed of 36 specific recommendations that the contributors should focus to improve their tests. The most frequent specific recommendations are: *unit test*, *specific snippet*, *end-to-end test*, *edge case*, and *expected exception*.

3.2.4 RQ4: How are the test reviews solved by contributors?

Table 3.4 presents that solved test reviews are rather frequent, especially those not involving any kind of doubts (related to the request or the implementation). When responding to a test review, a contributor, *i.e.,* the pull request author, might implement the requested test, ask for help, reject the request, or ignore/abandon it. We group those responses based on whether the contributor solved the request, and assess the frequency of both solved and unsolved cases.

Table 3.4: Contributor's feedback, their categories, and frequencies

Category Classes	Frequency
Solved	**252**
Solved without doubts	223
Solved with discussion	29
Unsolved	**72**

In addition, we compare those frequencies taking into account the different test request categories, *i.e., add test* and *change existing test*. Figure 3.10 illustrates the prevalence of *solved* cases and presents in ascending order the request categories most frequently solved. The least solved *add test* requests have classes as their SUT (55% of the cases are *unsolved*). Second, we have the *improve existing tests* which are *unsolved* in 42% of the cases. Next, we find *cover a method*, *cover a branch*, *unspecified SUT*, *cover a statement*, and *cover a case* ranging from 63% to 81% *solved* cases respectively. Lastly, requests for *refactoring* are the second most frequently solved *change existing test* (92%) and *fix* is completely solved (100%).

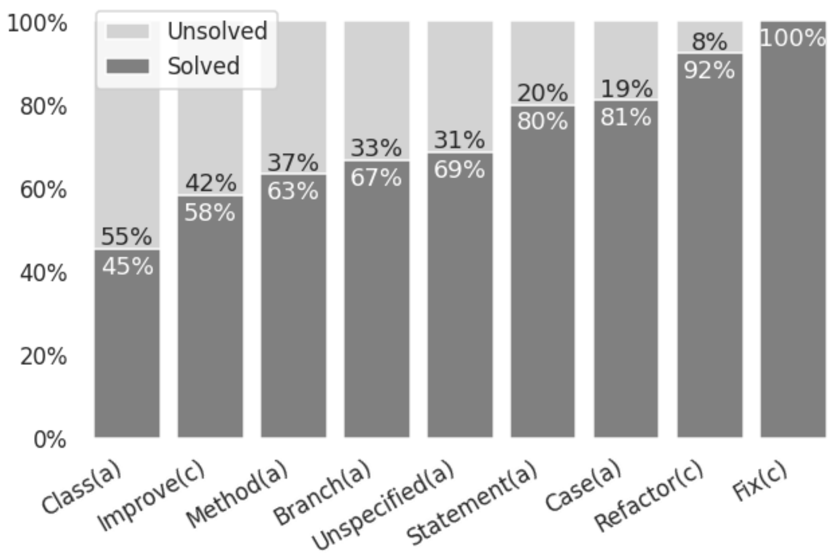


Figure 3.10: Solved ratio per request category ("c": *change existing tests*; "a": *add test*)

Finally, we assess the relationship between the number of recommendations in a request and the presence of a solving response. In Figure 3.11, we see that 75% of the *unsolved* requests have at most one recommendation and at least 25% have no recommendations. Furthermore, we find that 75% of the *solved* requests have at least one recommendation and 25% have more than two recommendations. Nevertheless, solved and unsolved test reviews have one recommendation on the median. Moreover, we apply both the Mann-Whitney test and the Cohen's d effect-size and we find a statistical difference between both groups and a low effect-size (-0.28).

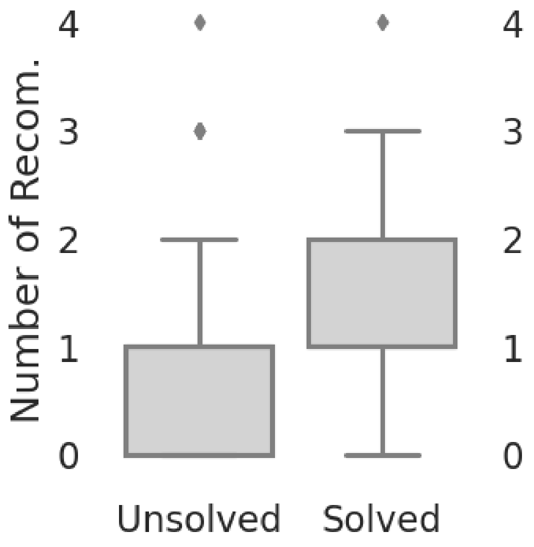


Figure 3.11: Distribution of recommendations among *unsolved* (left) and *solved* (right) test reviews

Summary: Overall, we find that test reviews are likely to be solved. We find that contributors solve test *fixes* and *refactoring* more frequently than other changes. Moreover, reviews with more recommendations are more likely to be *solved*.

3.3 Discussion and Implications

Novel empirical data on test review. In RQ1, we find that test reviews are indeed performed on popular programming languages and projects. Overall, we detect 11,836 test reviews in 5,421 open-source repositories. In RQ2, we detect that test reviews contains requests to add tests and change existing tests. In RQ3, we find seven broad categories of recommendations: *test scope*, *tool support*, *test scenarios*, *goal/purpose*, *refactoring*, *bad practices*, *fixtures*, and *miscellaneous*. Those broad categories are composed of 36 specific recommendations that the contributors should focus to improve their tests. Lastly, in RQ4, we detect that test reviews are likely to be solved. Moreover, reviews with more recommendations are more likely to be *solved*.

Test reviews with fine-grained details are more likely to be solved. We find that 80% of test reviews with fine-grained details, like single statement and cases, are solved, while less than half of the test reviews with class-level SUT are solved. In contrast, reviews targeting a coarse-grained SUT, like a class or method, are less likely to receive a test contribution. Thus, instead of one large SUT, reviewers should split and make multiple test reviews with smaller SUTs. Also in this context, we find that contributors solve test *fixes* and *refactoring* more frequently than other changes.

Focus on test scope and test scenarios. When providing recommendations, reviewers are more likely to focus on test scope and test scenarios (199 out of 370 recommendations belong to those categories). And, even though academia reached a consensus on the distribution of tests in the *test scope* range (*unit tests*, followed by *integration tests*, and then *end-to-end tests*[3]), that is not what we see in test reviews. While, as expected, *unit test* is the most commonly asked *test scope*, *integration* is surprisingly the least requested *test scope* representing less than 10% of their occurrences.

Just like any source code, tests are refactored. We find more than one-third of the test reviews that ask to *change existing tests* aim for refactoring. We also identified six commonly requested refactoring types in test reviews. Two of them target the SUT (*i.e., increase testability* and *extract member*) while the others target the test code (*i.e., rename test*, *improve assert message*, *move test*, and *merge test*).

[3] https://martinfowler.com/articles/practical-test-pyramid.html

Not only why, but what and how. Recent studies found absent rationale in pull requests and code reviews leads to higher rejection rate, confusion, and evaluation latency [8, 60, 61, 84], thus change requests should be properly motivated for a higher probability of being solved. Besides, we assess how reviewers use recommendations to better describe what is being requested and how to implement it. We find contributors are more willing to solve a test review when it contains at least one recommendation. Therefore, researchers should take into account the combination of both motivation and the level of detail of test reviews to better understand the reasons behind developer turnover.

3.4 Threats to Validity

Classification: Given the nature of a request recommendation and how we catalog it, it is expected that our list does not comprehend all existing test recommendations, but the most common ones. To mitigate that effect, we collected a great amount of data by choosing the latest version of the GHTorrent and making a statistically representative sample of it.

Precision goal: Before querying the GHTorrent dataset we set a fair precision goal of 85%. Then we incrementally built our query until its output exceeded the desired goal. Lastly, our final regular expression is the result of five iterations of this process, followed by a manual evaluation. Nevertheless, there might be missing cases as a consequence of our inability to measure the query recall, due to the lack of similar studies or datasets.

Offline context: Beyond the GitHub pull requests, some teams may use other communication means [85, 86] or even meet in person, which limits our capability of identifying request recommendations and response types. In practice, this may lead to over-counting ignored test requests and losing contextual information (references to unavailable knowledge or discussions). Since other platforms may have ad-hoc, thus hard to inspect, association with the code review discussion, we opted to only distinguish between ignored and rejected test requests in our public dataset. We acknowledge the importance of further developing that association study by considering test reviews to better understand the frequency they are ignored.

Generalizability: We analyzed hundreds of test reviews provided by open-source projects. Such projects are diverse, except they are all public on GitHub, indexed by GHTorrent [15], and contain at least one test review. In addition, although the stakeholders could be socially diverse, such characteristics, *e.g.,* gender [87, 88], ethnic group [89], culture [90], and personality trait [91], may not be available in their GitHub profile, which imposes challenges to measuring social bias. Therefore, our findings — as usual in empirical software engineering — may not be directly generalized to other contributors, that belong to social minorities, or to other systems, such as commercial ones with closed source, hosted in other open-source repositories (other than GitHub), and not indexed by GHTorrent.

3.5 Final Remarks

In this chapter, we presented the main study of this dissertation to explore how developers review tests on GitHub. We provided an empirical study to assess 11,836 test reviews from 5,421 open-source projects on GitHub. We also manually analyzed a sample 324 test reviews to better understand their content. Finally, based on our results, we discussed implications for researchers and practitioners.

Chapter 4

A Tool to Characterize Test Method Quality

In this chapter, we propose a tool for measuring the quality of individual test methods. We explain and validate the tool with an empirical study. This chapter is organized as follows: Section 4.1 presents our tool's approach, Section 4.2 details the study design, Section 4.3 reveals the results, Section 4.4 features discussions and implications, and Section 4.5 discusses this study's threats to validity.

4.1 Mutation Testing at Method Level

4.1.1 Test Method Mutation

To address the discussed limitations, we propose a five steps approach, as detailed in Figure 4.1. The first three steps are similar to traditional mutation testing: (1) run the test suite and collects the expected output; (2) parse the project and apply mutation operators; (3) the resulting mutants are separately tested by the test suite; (4) the result of *each* executed test method on *each* covered mutant is collected as the obtained output; and (5) the obtained output is compared to the expected result and the scores are computed for *each* test method. This approach is implemented by extending the mutation testing tool PIT tool [16] and is publicly available at `https://github.com/victorgveloso/Detailed-CSV-Report-PITest`.

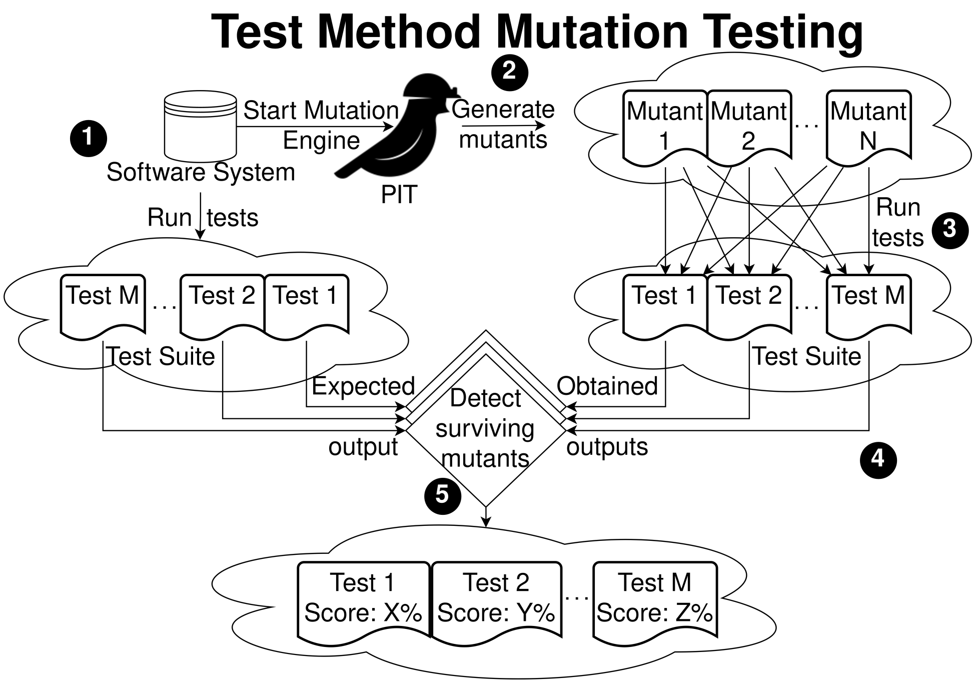
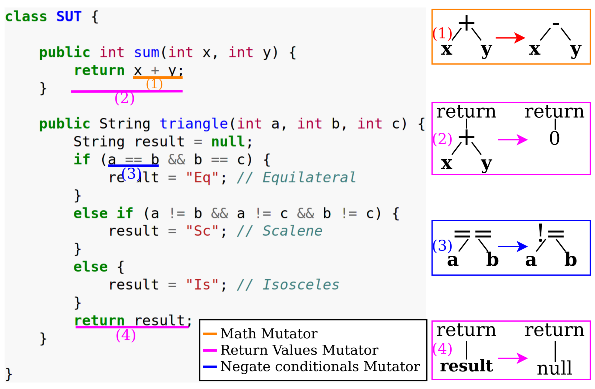


Figure 4.1: Overview of our approach

The score for a test method *test* is the ratio of mutants killed by the *test* and the total number of mutants the *test* covers. Given a test method, its *survived* mutants set is formed by the successful runs and its *killed* mutants by failures and errors. Notice that we do not include the *time-out* set to avoid noise in the collected output, which degrades the ability to define test methods quality.

4.1.2 Example: Computing Test Method Scores

In Figure 4.2 (which is repeated for simplicity), we note that 5 out of the 9 test methods have a mutation score of 100% (column "TM Score"), two have a score of 50%, and two have a score of 0%. Both `testTriangle5()` and `testTriangle6()` scores are 0%, suggesting they have less quality. Indeed, their assertions (*i.e.,* `assertNotEquals`) are the weakest in the test suite.



```
class SUT {

    public int sum(int x, int y) {
        return x + y;
    }

    public String triangle(int a, int b, int c) {
        String result = null;
        if (a == b && b == c) {
            result = "Eq"; // Equilateral
        }
        else if (a != b && a != c && b != c) {
            result = "Sc"; // Scalene
        }
        else {
            result = "Is"; // Isosceles
        }
        return result;
    }
}
```

(1)
(2)
(3)
(4)
Math Mutator
Return Values Mutator
Negate conditionals Mutator
(1) + x y → - x y
(2) return + x y → return 0
(3) == a b → != a b
(4) return result → return null

SUT Method	TM Name	TM Score	Assertion	Covered Mutants	Obtained Result	Status
sum	testSum1	100%	assertEquals(sum(4,5),9)	(1)	-1	killed
				(2)	0	killed
	testSum2	100%	assertEquals(sum(6,-5),1)	(1)	11	killed
				(2)	0	killed
	testSum3	100%	assertEquals(sum(-2,-4),-6)	(1)	2	killed
				(2)	0	killed
triangle	testTriangle1	100%	assertEquals(triangle(1,2,2),“Is”)	(3)	“Eq”	killed
				(4)	null	killed
	testTriangle2	50%	assertEquals(triangle(1,2,3),“Sc”)	(3)	“Sc”	survived
				(4)	null	killed
	testTriangle3	100%	assertEquals(triangle(1,1,1),“Eq”)	(3)	“Is”	killed
				(4)	null	killed
	testTriangle4	50%	assertNotEquals(triangle(1,2,2),“Eq”)	(3)	“Eq”	killed
				(4)	null	survived
	testTriangle5	0%	assertNotEquals(triangle(1,2,3),“Eq”)	(3)	“Sc”	survived
				(4)	null	survived
	testTriangle6	0%	assertNotEquals(triangle(1,1,1),“Sc”)	(3)	“Is”	survived
				(4)	null	survived
Test Suite Mutation Score:						100%

Figure 4.2: Score computation example in mutation testing inspired by [62] (“TM”: Test Method)

4.2 Study Design

4.2.1 Selecting the Software Systems

We collect the top 15 Java repositories from GitHub (in terms of the star metric [92, 93]) and the Apache Commons Lang. Next, for each project, we clone the latest master branch version and manually configure the extended PIT [16] via their build configuration file. We discard some projects due to PIT accusing their test suite of not being green, *i.e.,* some test cases did not pass during the test coverage evaluation phase. In addition, we detect that some of the projects are multi-module, *i.e.,* Okhttp, Retrofit, and ZXing. That is, they can be seen as a set of sub-projects (each one with its configuration files, build files, etc.) In those cases, we restrict the analysis to the core module and we discard the projects that do not explicitly specify a core module.[1] The five remaining projects are highly active and their size ranges from 35.9KLOC (Retrofit) to 310.8KLOC (RxJava).

4.2.2 Running the Mutation Testing Tool

After selecting the target projects and enabling all mutation operators supported by PIT, we start the mutation testing execution phase. Table 4.1 summarizes this analysis: in total, PIT detected 18,321 test cases in the five projects. Overall, it generated 55,427 mutants, which resulted in 16,149,383 mutant executions. The mutation scores are overall

[1]Core modules are the ones named with *core* or the project’s name.

high, ranging from 73% (Okhttp) to 86% (Commons Lang). For comparison purposes, we also present the coverage values in the last column. As expected [62], the coverage values are frequently higher than the mutation score.

Table 4.1: Projects test quality overview

Project	Tests	Mutants	TM Runs	Score	Cov.
Commons Lang	3,668	13,517	1.243M	86%	95%
RxJava	12,145	22,342	6.368M	85%	100%
ZXing	408	11,918	1.121M	75%	94%
Retrofit	337	883	0.154M	75%	51%
Okhttp	1,763	6,767	7.261M	73%	86%

4.2.3 Selecting the Test Methods

The next step is to select the test methods to be analyzed. To be selected for this study, test methods must: (1) contain a `@Test` annotation or a name prefixed by *test*, (2) not rely on anonymous classes, (3) not contain neither `@Ignore` nor `@Disabled` annotations, and (4) have a mutation score computable by PIT, *i.e.,* it covers at least one mutant. Next, we collect the mutation score of the test methods individually, extract the top-100 methods and bottom-100 methods in terms of mutation score, and randomly select 100 methods.

4.2.4 Research Questions

RQ5 (Quality)

In this RQ, we investigate high and low-quality test methods' code and evolution by computing six metrics: three evolutionary metrics (from PyDriller [94]) and three test-related (from *tsDetect* [95]). Those are largely adopted tools in the testing and software mining literature [71, 96–99].

Test Size. We assess the size of the test methods in terms of *source lines of code* (SLOC). Big test methods are heavy and hard to read [95].

Test Quality. We assess three metrics specific to test methods [95]. *Number of exceptions* measures the amount of exception-related code structures. We compute the *number of* *bad asserts* (*i.e.,* asserts without an explanation) present in test methods [99]. Lastly, *magic numbers* are direct references to numbers in the tests.

Contributors. We assess *developers' expertise* as the ratio of commits they authored in the target project and *number of contributors* is how many distinct developers changed each test method.

Modifications. Evolution is an important aspect of any source code and tests are not different. We analyze the number of changes (commits) in the test methods to understand their stability.

Rationale. Assessing to what extent high and low-quality test methods are associated with code evolution and static metrics is relevant for both practitioners and researchers. Practitioners may consider using static metrics which are cheaper in terms of space and time as a proxy of test quality. On the research side, this may support the prediction of test method quality [100] based on both metrics.

RQ6 (Test Smells)

This RQ assesses the impact of test smells (*i.e.,* sub-optimal design choices made when developing tests [101]) on test methods in terms of mutation score. Like RQ5, we rely on *tsDetect* [95] and analyze the latest version of the repositories' master branch. We assess ten test smells [95, 102]: Assertion Roulette, Duplicate Assert, Conditional Test Logic, Dependent Test, Sleepy Tests, Sensitive Equality, General Fixture, Magic Number Test, Exception Catching Throwing, and Unknown Tests. We select the top 10 most consolidated test smells and discard the ones: unrelated to test methods (e.g., Constructor Initialization), debatable in the literature (e.g., Mystery Guest), and infrequent in our dataset (e.g., Default Test).

Rationale. Recent studies focus on test smells [74], their impact on defect and change-proneness [73], and co-occurrence with code smells [72]. Still, their relationship with mutation score is unclear.

4.3 Results

4.3.1 RQ5: What are the code and evolutionary characteristics of high-quality test methods?

Table 4.2 summarizes the metric values for the best (top-100), random (100-random), and worst (bottom-100) methods. We apply the Mann-Whitney test at *alpha value* = 0.05 and the Cohen's d effect size between the best and worst test methods (column "Best vs. Worst"). We find a statistically significant difference in all metrics, with at least a very small effect. Next, we highlight some differences.

Number of lines of code. The best test methods are only slightly smaller than the worst ones (9 vs. 10, very small effect size).

Number of bad asserts. As most asserts are written without any explanation, this metric can be seen as a proxy of *number of asserts*. High-quality test methods have, on average, more asserts (3.7) than low-quality ones (1.6), but the difference is only small.

Number of modifications. The best test methods are only slightly less modified than the worst ones (mean 3.3 vs. 3.9, small effect).

Summary: There is no major difference between the *worst* group and the *best* group. The maximum effect-size is small and five out of seven metrics yield the same median.

Table 4.2: Metrics overview ($\tilde{\eta}$: median; Rnd: Random; N: Negligible; VS: Very Small; S: Small; H: Huge)

Metric	Best $\tilde{\eta}$	Rnd $\tilde{\eta}$	Worst $\tilde{\eta}$	Best vs. Worst p-value	Best vs. Worst effect-size
Nº of lines of code	10	10	9	< 0.05	VS
Nº of bad asserts	2	1	1	< 0.05	S
Nº of exceptions	0	0	0	< 0.05	S
Nº of magic numbers	0	0	0	< 0.05	VS
Nº of contributors	2	2	2	< 0.05	VS
Nº of modifications	3	3	3	< 0.05	S
Developer expertise	0.1	0.1	0.2	< 0.05	VS
Score	0.9	0.6	0	< 0.05	H

4.3.2 RQ6: What test smells are prevalent in high-quality test methods?

Figure 4.3 compares the presence of test smells on both high and low-quality test methods. Sleepy Tests, often related to non-determinism and flaky tests [103, 104], only occur in the worst group. Next, we see that 76% of General Fixture cases affect low-quality test methods. Moreover, 68% of Unknown Test happen in low-quality test methods. Finally, Conditional Test Logic and Exception Catching Throwing are also more likely to happen in low-quality test methods, however, the difference is smaller (58% vs. 42% and 54% vs. 46%, respectively). On the other hand, we see some test smells occurring more often in the best test methods, *e.g.,* Magic Number Test, Assertion Roulette, and Duplicate Assert. Those test smells are very controversial, for instance, Assertion Roulette represents test methods with more than one assert without explanation/message, which is a common practice in software testing. Indeed, those test smells are more related to test readability and do not directly affect the ability of the test to catch bugs.

Summary: Critical test smells are overconcentrated in the *worst* group, while maintainability-related are prevalent in the *best* group.

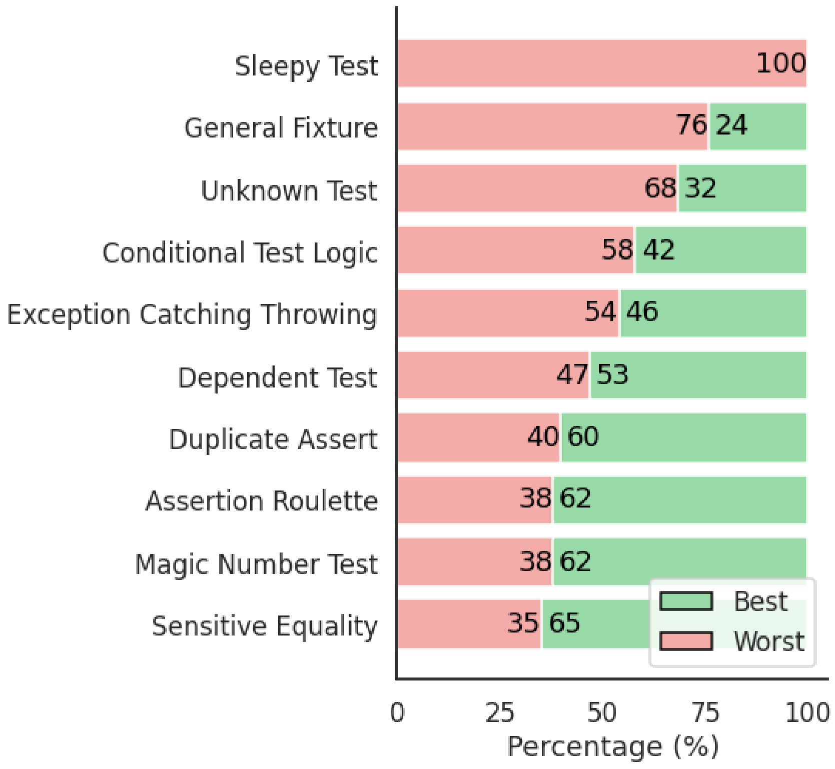


Figure 4.3: Prevalence of test smells

4.4 Discussion and Implications

Code and evolutionary characteristics. It is conventional wisdom that test methods should be small and non-complex to improve their maintainability [105]. However, we lack empirical data showing the real benefits of having those factors. We find no major differences between high-quality and low-quality test methods in terms of size, number of asserts, and modifications. This opens room for novel research to better understand the differences between high and low-quality test methods.
Test smells. Recent studies show that test smells may decrease the understandability and maintainability of the test suites [72, 73, 106, 107], despite practitioners do not perceive test smells as actual problems [72, 74]. In this study, we find that low-quality test methods are more likely to include critical test smells. For example, low-quality test methods are over-concentrated on Sleepy Test, General Fixture, and Unknown Test. On the other hand, high-quality test methods have less critical test smells, which are related to test readability, like Magic Number Test and Assertion Roulette. Thus, practitioners in charge of maintaining test suites should be aware that the presence of some test smells is associated with the test suite's ability in catching real bugs.

4.5 Threats to Validity

Timed out tests. PIT implements heuristics to identify mutants suffering from infinite loops. We discarded *time-out* occurrences from the test method score's formula to prevent noise in the score.

Failing test suite. Mutations to static members [108] and tests depending on a specific execution order may be falsely accused of having a non-green test suite. Solved by forcing PIT to execute mutants in separate processes and discarding the failing projects.

Anonymous classes. We discard test methods using anonymous classes, due to *tsDetect* tool [95] incompatibility.

Generalization. We analyzed thousands of test methods provided by open-source Java projects. However, our findings may not be directly generalized to other systems, as commercial ones with closed source and implemented in other languages.

4.6 Final Remarks

In this chapter, we proposed an empirical study to assess the quality of *test methods* by relying on our extension of PIT that achieves mutation testing at the method level. We show empirical evidence that there are no major differences between high-quality and low-quality test methods in terms of size, number of asserts, and modifications. Low-quality test methods are over-concentrated on critical test smells, while high-quality test methods are likely to contain less important ones.

Chapter 5

Conclusion

This chapter provides the final remarks of this master dissertation. In Section 5.1, we present the overview of our two studies and their contributions. And in Section 5.2, we propose future works.

5.1 Overview and Contributions

In this master dissertation, we presented an empirical study on test reviews where reviewers request contributors to add or change test methods. Specifically, we investigated four main questions: (1) How frequently do reviewers ask contributors for tests? (2) What do reviewers ask contributors for? (3) Which recommendations are frequently found in test requests? (4) How are the test reviews solved by contributors? In addition, we extended PIT, the state-of-the-art mutation testing framework enabling method-level reports. Then, we conducted an exploratory study to determine how mutation testing at the method level relates to static test quality metrics, such as number of lines of code, number of asserts, and test smells. For that we proposed two extra research questions: (5) What are the code and evolutionary characteristics of high-quality test methods? (6) What test smells are prevalent in high-quality test methods? We summarize the results and the major contributions of this master dissertation in the following subsections.

5.1.1 How Developers Review Tests on GitHub

We conducted an empirical study to analyze 11,836 test reviews on GitHub in which reviewers ask contributors to add and change tests within pull request discussions from 5,421 open-source projects. We assessed a sample of 324 test reviews, relying on a set of 36 extracted recommendations, to learn what information is available when contributors implement the required tests. Furthermore, we studied the contributors' responses, whether they further contribute to the pull request or not, and what factors influenced them. Based on our results, we provided a list of implications for researchers and practitioners. We presented empirical evidence that testing is an essential practice when contributing to open-source projects as reviewers usually request test development in case they perceive insufficient code coverage. In this context, the main findings of this study are as follows:

- Most contributors facing challenges completing the requested tasks do not reply to the reviewer and abandon the contribution, while a few others ask for the request clarification.
- Contributors are more willing to solve a *test review* that contains at least one *recommendation.*
- Coarse-grained SUT, *e.g.,* classes, are less likely to receive test contribution.
- *Refactoring* comprises more than one-third of the test reviews requesting *changes to existing tests*, particularly *refactoring* that targets the SUT and test-specific ones.
- The prevailing test reviews target the *test scope* and the *test scenario.*
- Integration test is the least mentioned *test scope*, comprising only 10% of the test reviews mentioning a *test scope.*

5.1.2 A Tool to Characterize Test Method Quality

We proposed an empirical study to assess the quality of *test methods* by relying on our extension of PIT that achieves *mutation testing at the method level.* We show empirical evidence that there are no major differences between high-quality and low-quality test methods in terms of size, number of asserts, and modifications. Low-quality test methods are over-concentrated on critical test smells, while high-quality test methods are likely to contain less important ones.

5.2 Future Work

Assess the quality of requested test methods. Even though the majority of test reviews are solved, we are not sure about the quality of the added or changed test methods. In this context, researchers can use the tool proposed in this study to compare the quality of spontaneous testing to requested testing.

Other sources of asynchronous and synchronous communications. GitHub is the most popular Git platform that includes social features, such as starring, following, issue tracking, and pull requests. On the other hand, there are many other social platforms extensively studied in the literature, *e.g.,* Gerrit, Gitter, Slack, GitLab, and SourceForge. And some teams may use multiple platforms simultaneously, so requests we interpreted as ignored in a platform could have been addressed in another one. Hence, our test review dataset can be further expanded by introducing projects from those platforms.

Consider more static and dynamic quality metrics. We found no major difference between high-quality and low-quality test methods in terms of the selected metrics (*e.g.,* size, number of asserts, and modifications). However, there are many other static metrics (*e.g.,* assertion types, input type, distance between test and SUT, cohesion, coupling, complexity, code smells, and many social metrics) and dynamic metrics (*e.g.,* stack-trace, dependency hierarchy, and all metrics from the code coverage family of metrics) that are candidates for analysis in future research.

Assess how test method quality relates to defect presence. Mutation testing is traditionally used to measure the quality of test suites. Many studies applied mutation testing on test suites affected with real defects to validate that technique and others evaluated whether mutation testing can positively impact test code readability. Extracting test methods from known defects datasets and applying our *method-level mutation testing* approach can highlight its ability to detect defects.

Survey on how perceived test method quality relates to our metric. Beyond the association of low test-method score and test smell occurrence, one can assess whether testers perceive a lack of maintainability on low-quality test methods. Surveying testers about their perception of either test methods identified as low-quality or high-quality, by our metric, can shed even more light on the relationship between test thoroughness and test maintainability.

Bibliography

[1] Marco Tulio Valente. “Engenharia de software moderna”. In: *Princípios e Práticas para Desenvolvimento de Software com Produtividade* 1 (2020).

[2] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software engineering at google: Lessons learned from programming over time.* O’Reilly Media, 2020.

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

[4] Yu Huang, Denae Ford, and Thomas Zimmermann. “Leaving My Fingerprints: Motivations and Challenges of Contributing to OSS for Social Good”. In: *International Conference on Software Engineering.* 2021, pp. 1020–1032.

[5] Amanda Lee, Jeffrey C. Carver, and Amiangshu Bosu. “Understanding the Impressions, Motivations, and Barriers of One Time Code Contributors to FLOSS Projects: A Survey”. In: *International Conference on Software Engineering.* 2017, pp. 187–197.

[6] Marco Gerosa, Igor Wiese, Bianca Trinkenreich, Georg Link, Gregorio Robles, Christoph Treude, Igor Steinmacher, and Anita Sarma. “The Shifting Sands of Motivation: Revisiting What Drives Contributors in Open Source”. In: *International Conference on Software Engineering.* 2021, pp. 1046–1058.

[7] Adam Alami, Marisa Leavitt Cohn, and Andrzej Wąsowski. “Why Does Code Review Work for Open Source Software Communities?” In: *International Conference on Software Engineering.* 2019, pp. 1073–1083.

[8] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. “Code Review Quality: How Developers See It”. In: *International Conference on Software Engineering.* Austin, Texas: ACM, 2016, pp. 1028–1038.

[9] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. “Wait for It: Determinants of Pull Request Evaluation Latency on GitHub”. In: *Working Conference on Mining Software Repositories.* 2015, pp. 367–371.

[10] 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: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* Lake Buena Vista, FL, USA: ACM, 2018, pp. 201–212.

[11] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. "Studying the Impact of Adopting Continuous Integration on the Delivery Time of Pull Requests". In: *International Conference on Mining Software Repositories*. Gothenburg, Sweden: ACM, 2018, pp. 131–141.

[12] Mohammad Masudur Rahman and Chanchal K. Roy. "Impact of Continuous Integration on Code Reviews". In: *International Conference on Mining Software Repositories*. 2017, pp. 499–502.

[13] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. "A Study on the Interplay between Pull Request Review and Continuous Integration Builds". In: *International Conference on Software Analysis, Evolution and Reengineering*. 2019, pp. 38–48.

[14] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. "The Silent Helper: The Impact of Continuous Integration on Code Reviews". In: *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2020, pp. 423–434.

[15] Georgios Gousios. "The GHTorrent dataset and tool suite". In: *Working Conference on Mining Software Repositories*. San Francisco, CA, USA, 2013, pp. 233–236.

[16] PIT Mutation Testing. https://pitest.org. Nov. 2020.

[17] Victor Veloso and Andre Hora. *How Developers Review Tests in GitHub?* Zenodo, July 2022. URL: https://doi.org/10.5281/zenodo.6792932.

[18] Dong Wang, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. "Automatic patch linkage detection in code review using textual content and file location features". In: *Information and Software Technology* 139 (2021), p. 106637.

[19] Matheus Paixao, Jens Krinke, Donggyun Han, and Mark Harman. "CROP: Linking Code Reviews to Source Code Changes". In: *International Conference on Mining Software Repositories*. Gothenburg, Sweden: ACM, 2018, pp. 46–49.

[20] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. "CORE: Automating Review Recommendation for Code Changes". In: *International Conference on Software Analysis, Evolution and Reengineering*. 2020, pp. 284–295.

[21] Victor Veloso and Andre Hora. *Characterizing high quality test methods, MSR'22*. Zenodo, June 2021. URL: https://doi.org/10.5281/zenodo.4987677.

[22] Nicole Davila and Ingrid Nunes. "A systematic literature review and taxonomy of modern code review". In: *Journal of Systems and Software* 177 (2021), p. 110951.

[23] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley K. G. Assunção, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. “Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study”. In: *International Conference on Mining Software Repositories*. 2021, pp. 471–482.

[24] Panyawut Sri-iesaranusorn, Raula Gaikovina Kula, and Takashi Ishio. “Does Code Review Promote Conformance? A Study of OpenStack Patches”. In: *International Conference on Mining Software Repositories*. 2021, pp. 444–448.

[25] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenilio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. “How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study”. In: *International Conference on Software Maintenance and Evolution*. 2020, pp. 511–522.

[26] DongGyun Han, Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, and Giovanni Rosa. “Does code review really remove coding convention violations?” In: *International Working Conference on Source Code Analysis and Manipulation*. 2020, pp. 43–53.

[27] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. “The Impact of Code Review on Architectural Changes”. In: *Transactions on Software Engineering* 47.5 (2021), pp. 1041–1059.

[28] Umme Ayda Mannan, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. “On the Relationship between Design Discussions and Design Quality: A Case Study of Apache Projects”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020, pp. 543–555.

[29] Anderson Uchôa. “Unveiling Multiple Facets of Design Degradation in Modern Code Review”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens, Greece: ACM, 2021, pp. 1615–1619.

[30] 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: *International Conference on Software Engineering*. Austin, Texas: ACM, 2016, pp. 1039–1050.

[31] Andrey Krutauz, Tapajit Dey, Peter C. Rigby, and Audris Mockus. “Do code review measures explain the incidence of post-release defects?” In: *Empirical Software Engineering* 25.5 (2020), pp. 3323–3356.

[32] Patanamon Thongtanunam and Ahmed E. Hassan. “Review Dynamics and Their Impact on Software Quality”. In: *Transactions on Software Engineering* 47.12 (2021), pp. 2698–2712.

[33] Gabriele Bavota and Barbara Russo. “Four eyes are better than two: On the impact of code reviews on software quality”. In: *International Conference on Software Maintenance and Evolution.* 2015, pp. 81–90.

[34] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. “Investigating code review quality: Do people and participation matter?” In: *International Conference on Software Maintenance and Evolution.* 2015, pp. 111–120.

[35] 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: *Mining Software Repositories.* 2015, pp. 168–179.

[36] Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. “The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems”. In: *Empirical Software Engineering* 26.2 (2021).

[37] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. “Modern Code Review: A Case Study at Google”. In: *International Conference on Software Engineering: Software Engineering in Practice.* ACM, 2018, pp. 181–190.

[38] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. “Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox”. In: *International Conference on Software Engineering: Software Engineering in Practice.* IEEE, 2021, pp. 348–357.

[39] Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart de Water. “Studying Pull Request Merges: A Case Study of Shopify’s Active Merchant”. In: *International Conference on Software Engineering: Software Engineering in Practice.* Gothenburg, Sweden: ACM, 2018, pp. 124–133.

[40] Amiangshu Bosu, Michaela Greiler, and Christian Bird. “Characteristics of Useful Code Reviews: An Empirical Study at Microsoft”. In: *Working Conference on Mining Software Repositories.* 2015, pp. 146–156.

[41] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. “Code Reviews With Divergent Review Scores: An Empirical Study of the OpenStack and Qt Communities”. In: *Transactions on Software Engineering* 48.1 (2022), pp. 69–81.

[42] Yuxia Zhang, Minghui Zhou, Klaas-Jan Stol, Jianyu Wu, and Zhi Jin. “How Do Companies Collaborate in Open Source Ecosystems? An Empirical Study of OpenStack”. In: *International Conference on Software Engineering.* 2020, pp. 1196–1208.

[43] Javier Luis Cánovas Izquierdo and Jordi Cabot. “On the analysis of non-coding roles in open source development”. In: *Empirical Software Engineering* 27.1 (2021), p. 18.

[44] Jean-Gabriel Young, Amanda Casari, Katie McLaughlin, Milo Z. Trujillo, Laurent Hébert-Dufresne, and James P. Bagrow. “Which contributions count? Analysis of attribution in open source”. In: *International Conference on Mining Software Repositories.* 2021, pp. 242–253.

[45] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. “Work Practices and Challenges in Pull-Based Development: The Integrator’s Perspective”. In: *International Conference on Software Engineering.* 2015, pp. 358–368.

[46] Daniel Izquierdo-Cortazar, Lars Kurth, Jesus M. Gonzalez-Barahona, Santiago Dueñas, and Nelson Sekitoleko. “Characterization of the Xen Project Code Review Process: An Experience Report”. In: *International Conference on Mining Software Repositories.* Austin, Texas: ACM, 2016, pp. 386–390.

[47] Thomas Bock, Claus Hunsen, Mitchell Joblin, and Sven Apel. “Synchronous development in open-source projects: A higher-level perspective”. In: *Automated Software Engineering* 29.1 (2021), p. 3.

[48] Alberto Bacchelli and Christian Bird. “Expectations, Outcomes, and Challenges of Modern Code Review”. In: *International Conference on Software Engineering.* IEEE, 2013, pp. 712–721.

[49] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. “Open Source Peer Review–Lessons and Recommendations for Closed Source”. In: *IEEE Software* 29.6 (2012), pp. 56–61.

[50] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. “Effects of Adopting Code Review Bots on Pull Requests to OSS Projects”. In: *International Conference on Software Maintenance and Evolution.* IEEE, 2020, pp. 1–11.

[51] Mairieli Wessel. “Enhancing Developers’ Support on Pull Requests Activities with Software Bots”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 2020, pp. 1674–1677.

[52] Yuqing Wang, Mika V. Mäntylä, Zihao Liu, and Jouni Markkula. “Test automation maturity improves product quality—Quantitative study of open source projects using continuous integration”. In: *Journal of Systems and Software* 188 (2022).

[53] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. “Would static analysis tools help developers with code reviews?” In: *International Conference on Software Analysis, Evolution, and Reengineering*. 2015, pp. 161–170.

[54] Fiorella Zampetti, Saghan Mudbhari, Venera Arnaoudova, Massimiliano Di Penta, Sebastiano Panichella, and Giuliano Antoniol. “Using code reviews to automatically configure static analysis tools”. In: *Empirical Software Engineering* 27.1 (2021), p. 28.

[55] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. “Effectiveness of Code Contribution: From Patch-Based to Pull-Request-Based Tools”. In: *International Symposium on Foundations of Software Engineering*. Seattle, WA, USA: ACM, 2016, pp. 871–882.

[56] Marcus Vinicius Bertoncello, Gustavo Pinto, Igor Scaliante Wiese, and Igor Steinmacher. “Pull Requests or Commits? Which Method Should We Use to Study Contributors’ Behavior?” In: *International Conference on Software Analysis, Evolution and Reengineering*. 2020, pp. 592–601.

[57] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. “Work Practices and Challenges in Pull-Based Development: The Contributor’s Perspective”. In: *International Conference on Software Engineering*. 2016, pp. 285–296.

[58] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. “When Testing Meets Code Review: Why and How Developers Review Tests”. In: *International Conference on Software Engineering*. ACM, 2018, pp. 677–687.

[59] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. “Confusion Detection in Code Reviews”. In: *International Conference on Software Maintenance and Evolution*. 2017, pp. 549–553.

[60] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. “Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies”. In: *International Conference on Software Analysis, Evolution and Reengineering*. 2019, pp. 49–60.

[61] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. “An exploratory study on confusion in code reviews”. In: *Empirical Software Engineering* 26.1 (2021), p. 12.

[62] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. “The Fuzzing Book”. In: Saarland University, 2019. URL: `https://www.fuzzingbook.org`.

[63] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. “Test-Driven Code Review: An Empirical Study”. In: *International Conference on Software Engineering*. 2019, pp. 1061–1072.

[64] Ruiyin Wen, Maxime Lamothe, and Shane McIntosh. “How Does Code Reviewing Feedback Evolve?” In: *International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2022.

[65] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. “Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down”. In: *International Conference on Software Engineering*. Vol. 2. 2015, pp. 27–28.

[66] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. “An empirical study of the impact of modern code review practices on software quality”. In: *Empirical Software Engineering* 21.5 (2016), pp. 2146–2189.

[67] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. “Communicative Intention in Code Review Questions”. In: *International Conference on Software Maintenance and Evolution*. 2018, pp. 519–523.

[68] Itir Karac, Burak Turhan, and Natalia Juristo. “A Controlled Experiment with Novice Developers on the Impact of Task Description Granularity on Software Quality in Test-Driven Development”. In: *Transactions on Software Engineering* 47.7 (2021), pp. 1315–1330.

[69] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *Transactions on Software Engineering* (2011), pp. 649–678.

[70] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. “Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality”. In: *International Conference on Software Maintenance and Evolution*. 2020, pp. 336–347.

[71] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. “How the experience of development teams relates to assertion density of test classes”. In: *International Conference on Software Maintenance and Evolution*. 2019, pp. 223–234.

[72] 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: *International Conference on Automated Software Engineering*. 2016, pp. 4–15.

[73] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. “On the Relation of Test Smells to Software Code Quality”. In: *International Conference on Software Maintenance and Evolution*. 2018, pp. 1–12.

[74] Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. “The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems”. In: *Empirical Software Engineering* 26.2 (2021), pp. 1–42.

[75] Michael Hilton, Jonathan Bell, and Darko Marinov. “A large-scale study of test coverage evolution”. In: *International Conference on Automated Software Engineering.* 2018, pp. 53–63.

[76] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. “Predictive Test Selection”. In: *International Conference on Software Engineering: Software Engineering in Practice.* 2019, pp. 91–100.

[77] Daniela S Cruzes and Tore Dyba. “Recommended steps for thematic synthesis in software engineering”. In: *International Symposium on Empirical Software Engineering and Measurement.* 2011, pp. 275–284.

[78] Luan P. Lima, Lincoln S. Rocha, Carla I. M. Bezerra, and Matheus Paixao. “Assessing exception handling testing practices in open-source libraries”. In: *Empirical Software Engineering* 26.5 (2021), p. 85.

[79] Diego Marcilio and Carlo A. Furia. “How Java Programmers Test Exceptional Behavior”. In: *International Conference on Mining Software Repositories.* 2021, pp. 207–218.

[80] Barbee E. Teasley, Laura Marie Leventhal, Clifford R. Mynatt, and Diane S. Rohlman. “Why software testing is sometimes ineffective: Two applied studies of positive test strategy.” In: *Journal of Applied Psychology* 79.1 (1994), pp. 142–155.

[81] Iflaah Salman, Pilar Rodríguez, Burak Turhan, Ayşe Tosun, and Arda Güreller. “What Leads to a Confirmatory or Disconfirmatory Behavior of Software Testers?” In: *Transactions on Software Engineering* 48.4 (2022), pp. 1351–1368.

[82] A. Page, K. Johnston, and B. Rollison. *How We Test Software at Microsoft.* Developer Best Practices. Pearson Education, 2008.

[83] Jung-Min Kim, Adam Porter, and Gregg Rothermel. “An empirical study of regression test application frequency”. In: *Software Testing, Verification and Reliability* 15.4 (2005), pp. 257–279.

[84] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. “Anti-patterns in Modern Code Review: Symptoms and Prevalence”. In: *International Conference on Software Analysis, Evolution and Reengineering.* 2021, pp. 531–535.

[85] Esteban Parra, Mohammad Alahmadi, Ashley Ellis, and Sonia Haiduc. “A comparative study and analysis of developer communications on Slack and Gitter”. In: *Empirical Software Engineering* 27.2 (2022), p. 40.

[86] Lin Shi, Xiao Chen, Ye Yang, Hanzhi Jiang, Ziyou Jiang, Nan Niu, and Qing Wang. “A First Look at Developers’ Live Chat on Gitter”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* Athens, Greece: ACM, 2021, pp. 391–403.

[87] Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. “Biases and Differences in Code Review Using Medical Imaging and Eye-Tracking: Genders, Humans, and Machines”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2020, pp. 456–468.

[88] Rajshakhar Paul, Amiangshu Bosu, and Kazi Zakia Sultana. “Expressions of Sentiments during Code Reviews: Male vs. Female”. In: *International Conference on Software Analysis, Evolution and Reengineering.* 2019, pp. 26–37.

[89] Reza Nadri, Gema Rodriguez-Perez, and Meiyappan Nagappan. “Insights Into Nonmerged Pull Requests in GitHub: Is There Evidence of Bias Based on Perceptible Race?” In: *IEEE Software* 38.2 (2021), pp. 51–57.

[90] Leonardo B. Furtado, Bruno Cartaxo, Christoph Treude, and Gustavo Pinto. “How Successful Are Open Source Contributions From Countries With Different Levels of Human Development?” In: *IEEE Software* 38.2 (2021), pp. 58–63.

[91] Rahul N. Iyer, S. Alex Yun, Meiyappan Nagappan, and Jesse Hoey. “Effects of Personality Traits on Pull Request Acceptance”. In: vol. 47. 11. IEEE, 2021, pp. 2632–2643.

[92] Hudson Borges, Andre Hora, and Marco Tulio Valente. “Understanding the Factors that Impact the Popularity of GitHub Repositories”. In: *International Conference on Software Maintenance and Evolution.* 2016, pp. 334–344.

[93] Hudson Silva and Marco Tulio Valente. “What’s in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform”. In: *Journal of Systems and Software* 146 (2018), pp. 112–129.

[94] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. “PyDriller: Python framework for mining software repositories”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2018, pp. 908–911.

[95] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. “TsDetect: An Open Source Test Smells Detection Tool”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2020, pp. 1650–1654.

[96] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. “The technical debt dataset”. In: *International Conference on Predictive Models and Data Analytics in Software Engineering*. 2019, pp. 2–11.

[97] Ayaan M Kazerouni, Clifford A Shaffer, Stephen H Edwards, and Francisco Servant. “Assessing incremental testing practices and their impact on project outcomes”. In: *Technical Symposium on Computer Science Education*. 2019, pp. 407–413.

[98] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn. “Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities”. In: *International Conference on Software Maintenance and Evolution*. 2020, pp. 523–533.

[99] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. “Investigating Severity Thresholds for Test Smells”. In: *International Conference on Mining Software Repositories*. 2020, pp. 311–321.

[100] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang. “Predictive Mutation Testing”. In: *Transactions on Software Engineering* 45 (2019), pp. 898–918.

[101] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[102] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. “Refactoring test code”. In: *International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*. 2001, pp. 92–95.

[103] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. “Understanding Flaky Tests: The Developer’s Perspective”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 830–840.

[104] Fabio Palomba and Andy Zaidman. “Does refactoring of test smells induce fixing flaky tests?” In: *International Conference on Software Maintenance and Evolution*. 2017, pp. 1–12.

[105] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[106] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. “An empirical analysis of the distribution of unit test smells and their impact on software maintenance”. In: *International Conference on Software Maintenance*. 2012, pp. 56–65.

[107] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. “Are test smells really harmful? An empirical study”. In: *Empirical Software Engineering* 20.4 (2015), pp. 1052–1094.

[108] Thomas Laurent and Anthony Ventresque. “PIT-HOM: an Extension of Pitest for Higher Order Mutation Analysis”. In: *International Conference on Software Testing, Verification and Validation Workshops*. 2019, pp. 83–89.