Characterizing High-Quality Test Methods: A First Empirical Study

Victor Veloso, Andre Hora
Department of Computer Science
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, Brazil
{victorveloso,andrehora}@dcc.ufmg.br

ABSTRACT

To assess the quality of a test suite, one can rely on mutation testing, which computes whether the overall test cases are adequately exercising the covered lines. However, this high level of granularity may overshadow the quality of individual test methods. In this paper, we propose an empirical study to assess the quality of test methods by relying on mutation testing at the method level. We find no major differences between high-quality and low-quality test methods in terms of size, number of asserts, and modifications. In contrast, high-quality test methods are less affected by critical test smells. Finally, we discuss practical implications for researchers and practitioners.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Mutation testing, Code quality, Software repository mining

ACM Reference Format:
Victor Veloso, Andre Hora. 2022. Characterizing High-Quality Test Methods: A First Empirical Study. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3529092

1 INTRODUCTION

Software testing is a key practice in software development. As a proxy of test quality, one can rely on code coverage or mutation analysis. Coverage measures the percentage of code that is covered by tests and is typically used to assess the test effectiveness [8, 12, 13]. However, it presents some limitations [5, 21, 34]. For example, one can have great coverage with no assert [34]. Another solution to assess test quality (and overcome such limitations) is mutation testing [5, 7, 14, 26, 34]. This technique injects mutations (artificial faults) into the code and checks if tests can detect (or "kill", in the mutation testing terminology) these mutations. The rationale is that if it fails to detect such mutations, it will miss real bugs [34].

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3529092

Code coverage and mutation score typically target the overall test suite effectiveness [6, 8, 9, 26]. However, this high level of granularity may overshadow test quality [12]. Mutation Testing at the method-level has proven useful for test suite reduction [27] and assessing test quality with it would be important to: (1) understand the characteristics of good (and bad) test methods, (2) provide novel empirical data at method level to differentiate both good and bad test methods, and (3) help to improve the quality of existing tests.

In this paper, we propose an empirical study to assess the quality of *test methods* by relying on mutation testing. We extend a state-of-the-art mutation testing tool [26] to analyze test methods and report mutation results at the method level. Then, 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 high-quality test methods:

RQ1: What are the code and evolutionary characteristics of high-quality test methods?
RQ2: What test smells are prevalent in high-quality test methods?

Overall, we find no major differences between high and low-quality test methods in terms of size, number of asserts, and modifications. In contrast, high-quality test methods are less affected by critical test smells.

Contributions: The contributions of this paper are twofold: (1) we provide an empirical study to characterize high-quality test methods and (2) we discuss implications for practitioners and researchers working on software testing.

Dataset: zenodo.org/record/4987677#.YMz8G5pKiA0

2 MUTATION TESTING IN A NUTSHELL

Test mutation technique assesses test effectiveness in four major steps (illustrated in Figure 1-left). First, the project test suite is executed and the results are stored as the *expected output*. Then, a mutation testing engine (*e.g.*, PIT [26]) 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 [34].

2.1 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 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 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.

2.2 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 outnumber 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 quality difference between the test methods, from the mutation analysis perspective.

3 MUTATION TESTING AT METHOD LEVEL

3.1 Test Method Mutation

To address the discussed limitations, we propose a five steps approach, as detailed in Figure 1-right. The first three steps are similar to the 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 [26] and is publicly available at: https://github.com/victorgveloso/Detailed-CSV-Report-PITest.

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.

3.2 Example: Computing Test Method Scores

In Figure 2, 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.

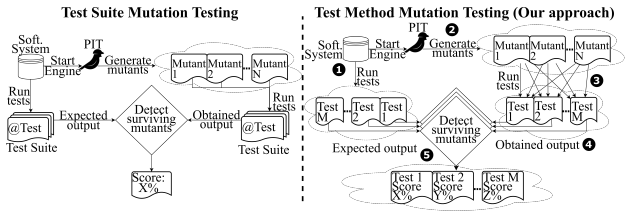


Figure 1: Traditional approach (left) vs. ours (right).

4 STUDY DESIGN

4.1 Selecting the Software Systems

We collect the top 15 Java repositories from GitHub (in terms of the star metric [3, 28]) and the Apache Commons Lang. Next, for each project, we clone the latest master branch version, manually configure the extended PIT [26] via their build configuration file, and discard the projects accused by PIT of having failing tests. The five remaining projects are highly active and their size ranges from 35.9KLOC (Retrofit) to 310.8KLOC (RxJava).

4.2 Running the Mutation Testing Tool

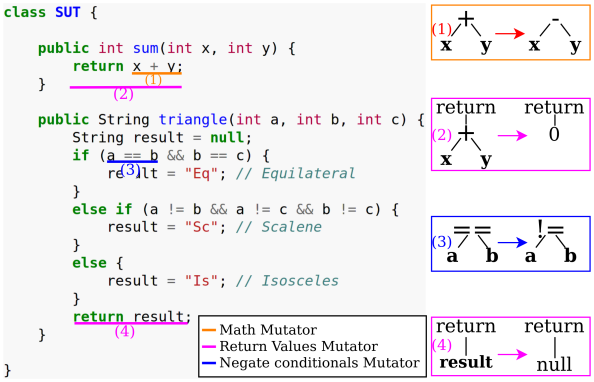
After selecting the target projects, we start the mutation testing execution phase. Table 1 summarizes this analysis: in total, PIT detected 18,321 test cases in the five projects. Overall, it generated 55,427 mutants, which resulted in 16,149,383 mutant executions. The mutation scores are overall high, ranging from 73% (Okhttp) to 86% (Commons Lang). For comparison purposes, we also present the coverage values in the last column. As expected [34], the coverage values are frequently higher than the mutation score.

Table 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%

4.3 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



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: Score computation example in mutation testing inspired by [34] ("TM": Test Method).

individually, extract the top-100 methods and bottom-100 methods in terms of mutation score, and randomly select 100 methods.

4.4 Assessing the Research Questions

4.4.1 RQ1 (Quality). In the first RQ, we investigate high and low-quality test methods' code and evolution by computing six metrics: three evolutionary metrics (from PyDriller [29]) and three test-related (from "tsDetect" [25]). Those are largely adopted tools in the testing and software mining literature [4, 15, 17, 23, 31].

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 [25].

Test Quality. We assess three metrics specific to test methods [25]. *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 [31]. 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 [35] based on both metrics.

4.4.2 RQ2 (Test Smells). This RQ assesses the impact of test smells (*i.e.,* sub-optimal design choices made when developing tests [20]) on test methods in terms of mutation score. Like RQ1, we rely on "tsDetect" [25] and analyze the latest version of the repositories' master branch. We assess ten test smells [25, 33]: 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 [24], their impact on defect and change-proneness [30], and co-occurrence with code smells [32]. Still, their relationship with mutation score is unclear.

5 RESULTS

5.1 RQ1 (Quality)

Table 2 summarizes the metric values for the 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).

Table 2: Metrics overview ($\bar{\eta}$: median; Rnd: Random; N: Negligible; VS: Very Small; S: Small; H: Huge).

Metric	Best $\bar{\eta}$	Rnd $\bar{\eta}$	Worst $\bar{\eta}$	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

5.2 RQ2 (Test Smells)

Figure 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 [10, 22], only occurs 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.

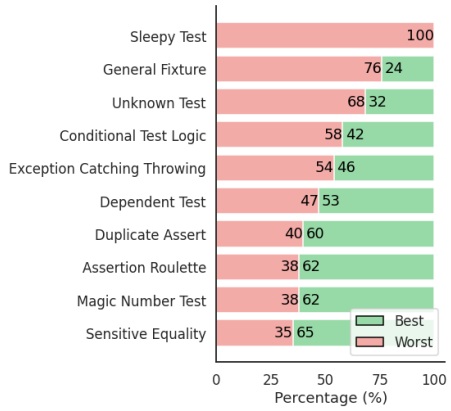


Figure 3: Prevalence of test smells.

6 DISCUSSION

Code and evolutionary characteristics. It is conventional wisdom that test methods should be small and non-complex to improve their maintainability [19]. 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 [1, 2, 30, 32], despite practitioners do not perceive test smells as actual problems [24, 32]. 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.

7 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 [16] 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 [25] 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.

8 RELATED WORK

Measuring test effectiveness through Mutation Testing is a largely studied topic with well-defined benefits and constraints [14]. Addressing those points, Jia *et al.* [14] summarize 390 studies in a public repository. On the other hand, Giovanni *et al.* [11] found that researchers and practitioners perceive existing metrics assist detecting low-quality test suites, but do not guarantee the high quality of a test suite. [11]. Catolino *et al.* [4] 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 [32], change-proneness and defect-proneness [30], and post-release defects [24]. Despite the richness of the test smell research topic, practitioners do not perceive test smells as actual problems [24, 32], 90% of test smells are never fixed, and fixing takes, on average, 100 days [32].

Hilton *et al.* assess the impact of finer granularity reports on some test coverage limitations. The authors describe how non-code changes impact test coverage and how finer granularity reports enable developers to better understand the quality of a specific change [12]. Despite being considerably cheaper than mutation testing, test coverage still has challenges when adopted in large-scale projects [18]. Challenges aggravated by the monorepo settings at Facebook, where test prioritization usage reduced infrastructure overhead [18].

9 CONCLUSION

We proposed an empirical study to assess the quality of *test methods* by relying on 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.

As future work, we plan to extend our dataset and to include more static and runtime metrics in the analysis to better assess test quality, *e.g.,* metrics related to the adopted assertions and the input data. Lastly, we plan to provide more qualitative analysis on the differences between high and low-quality test methods.

ACKNOWLEDGMENT

This research is supported by CAPES, CNPq, and FAPEMIG.

REFERENCES

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

[2] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.

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

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

[5] Code Coverage Best Practices. November, 2020. https://testing.googleblog.com/2020/08/code-coverage-best-practices.html.

[6] Codecov. November, 2020. https://codecov.io.

[7] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *International Symposium on Software Testing and Analysis*. 449–452.

[8] Coverage.py. November, 2020. https://coverage.readthedocs.io.

[9] Coveralls. November, 2020. https://coveralls.io.

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

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

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

[13] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 955–963.

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

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

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

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

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

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

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

[21] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. 2016. Will My Tests Tell Me If I Break This Code?. In *International Workshop on Continuous Software Evolution and Delivery*. 23–29.

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

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

[24] Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. 2021. The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems. *Empirical Software Engineering* 26, 2 (2021), 1–42.

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

[26] PIT Mutation Testing. November, 2020. https://pitest.org.

[27] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing Trade-Offs in Test-Suite Reduction. In *Foundations of Software Engineering*. 246–256.

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

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

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

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

[32] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An Empirical Investigation into the Nature of Test Smells. In *International Conference on Automated Software Engineering*. 4–15.

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

[34] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. Saarland University. https://www.fuzzingbook.org

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