Testability First!

Mohammad Ghafari
University of Bern
Bern, Switzerland
mohammad.ghafari@inf.unibe.ch

Markus Eggiman
University of Bern
Bern, Switzerland
markus.eggimann@students.unibe.ch

Oscar Nierstrasz
University of Bern
Bern, Switzerland
oscar.nierstrasz@inf.unibe.ch

Abstract—[Background] The pivotal role of testing in high-quality software production has driven a significant effort in evaluating and assessing testing practices. [Aims] We explore the state of testing in a large industrial project over an extended period. [Method] We study the interplay between bugs in the project and its test cases, and interview developers and stakeholders to uncover reasons underpinning our observations.

[Results] We realized that testing is not well adopted, and that *testability (i.e., ease of testing)* is low. We found that developers tended to abandon writing tests when they assessed the effort to be high. Frequent changes in requirements and pressure to add new features also hindered developers from writing tests.

[Conclusions] Regardless of the debates on test first or later, we hypothesize that the underlying reasons for poor test quality are rooted in a lack of attention to testing early in the development of a software component, leading to poor testability of the component. However, testability is usually overlooked in research that studies the impact of testing practices, and should be explicitly taken into account.

Index Terms—testability, test quality, testing practices

I. INTRODUCTION

Software engineers believe that high quality of software testing will lead to high quality in the software product. However, it is not so clear why the state of practice does not truly reflect this established wisdom, as software testing often suffers from low coverage and low effectiveness.

Monitoring the activity of about 2 500 developers over 2.5 years shows that half of developers do not test, most programming sessions end without any test execution, and software developers only spend a quarter of their time engineering tests, whereas they think they test half of the time [1]. Consequently, software systems are often released under-tested. What's worse, there are contradictory views on whether coverage criteria have a positive effect on finding faults. Some studies claim no correlation between unit test coverage and post-unit test defects [7], whereas a more recent study of open source projects shows that, on average, about 30% of defective methods are covered by JUnit tests, and the number of methods executed by a JUnit test is strongly related to that test uncovering a defect [11]. In order to improve the effectiveness of test suites, researcher suggest to focus on the strength of test oracles along with code coverage [12]. In the past years, there has been increasing effort to assess and improve the fault-detection capability of

test suites. The state-of-the-art technique is mutation testing, which checks the ability of a test suite to reveal artificially produced defects [9]. Nevertheless, this technique consumes enormous computing resources, thus hindering its application in practice. Researchers suggest to reduce the execution time of mutation testing, for example, by verifying the quality of the test cases for each individual method, instead of the overall test suite quality [14].

There is general consensus among software engineers that to have high quality tests software needs to be testable in the first place. However, to the best of our knowledge, the significance of *testability* in the quality of test suites has not received enough attention. There exist many definitions of software testability [5]. In this paper we use it to mean *"ease of testing"* in terms of effort needed to write tests that are both effective in revealing defects, and manageable in the face of changes.

We studied the interplay between bugs and test cases in a large industrial project in which a deliberate effort was made to improve testing quality partway through the project's lifetime. We affirm that many factors can influence and characterize the testability of the project, whereas state-of-the-art work that compares the effectiveness of various testing strategies in multiple projects often neglects to take into account the possibly different degrees of testability and other testing-related characteristics of these projects.

We observed both relatively low coverage and low effectiveness of tests. Many components were not tested because they were considered hard to test. It was common for bugs to be detected with the help of manual testing rather than by automated tests. Developers were reluctant to write tests for code that was subject to frequent changes. We observed that commitment from management and team dedication were important prerequisites to instill a healthy software testing culture. Finally, many useful tools and techniques have come from research efforts, yet they were often either not well-known, or high-quality implementations were not available, thus hampering their adoption in this company.

Digging deeper, we found that the root of the problem appeared to be that, when testing was not considered early in the development process, this led to software designs that made it more difficult to test the software *post hoc*. In contrast, when a deliberate management decision was taken to improve test coverage, this led to better software designs that were easier to test, and thus to improvements in software quality.

Regardless of existing debates on when to start automated testing in a software project, we hypothesize that establishment of clear testing policies applied throughout the entire development lifecycle, and early attention to testing of each component are crucial to ensure high testability of the code base, and thus high-quality tests. We therefore see a need to conduct new studies with multiple projects that share the same testing practices, and put similar effort into testing.

The rest of this paper is organized as follows. We introduce the case study in section II. We then present our key findings in section III, followed by a discussion in section IV. In section V we discuss related work, and conclude the paper in section VI.

II. CASE STUDY

The goal of this work was to learn about the state of testing in a large logistics company in Switzerland, which we will call *ABC*. In particular, we investigated *whether developers adopt a consistent testing practice, and if not, what are the reasons*. In the following we discuss the reasons behind choosing this case study, and then present our methodology.

A. Motivation

We study a digital platform for sending and receiving physical as well as electronic mail. We chose to study this platform for the following reasons: (i) it is a large real-world software system comprising 14 102 files with around half a million lines of Java and C# code, and about 15 500 issues, (ii) we have access to five years of development history from 2013 to 2018, easing the recovery of links among commits and issues, and (iii) we could approach over 40 developers and stakeholders for any matter about the project.

This platform, which we will call *XYZ*, started as a simple web application in 2011 and grew over time. The back-end is a large monolithic system that had its origins in that web application. There were automated unit tests for testing the business logic, but the coverage was generally low. As the project grew, it became increasingly difficult to manage. In the summer of 2017, the team decided to structure the whole system in a more modular way. New features should be added as modules, and these modules should be covered by unit tests. The team and the project management agreed to strive for 40%-80% code coverage of those new modules. Moreover, a team of testers would manually test the portal and the mobile apps, and manual regression tests would be executed after every development sprint.

Figure 1 shows the history of bugs in the *XYZ* project. The Y axis presents the cumulative number of bugs, and the X axis shows the time span of the project. The red dots show the total number of reported bugs at a given point in time, and the green dots show the number of resolved bugs. The gray vertical lines show the project releases.

Most of the time the green line follows closely behind the red line, which means that the total number of open bugs in the system at a given point in time remains more or less constant (on average 210 bugs). Whereas developers were continuously fixing bugs throughout the lifetime of the project, the green
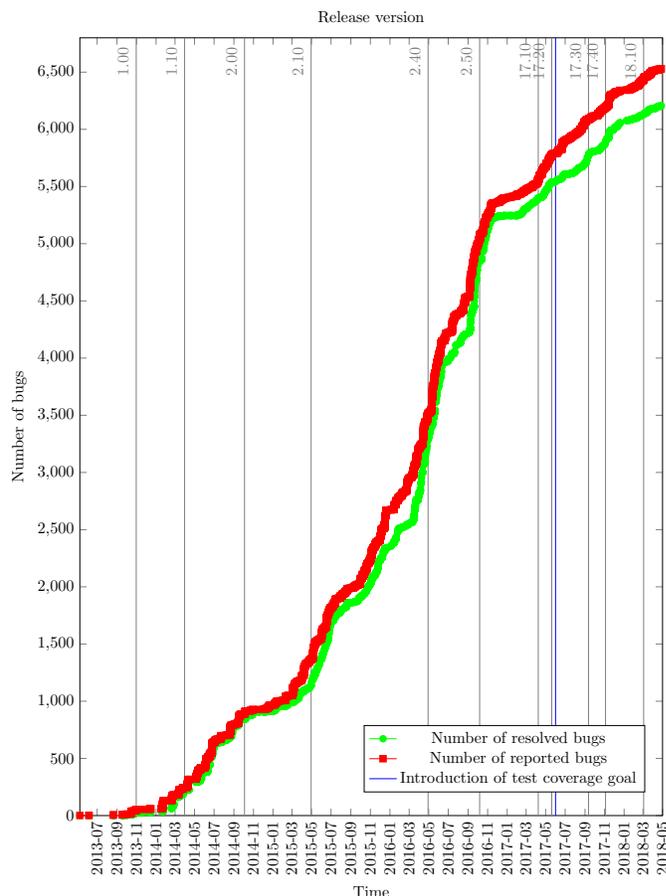

Fig. 1. Bug discovery and fixes over time

line and the red line take different paths after November 2016, and the number of open bugs slightly increases to 249 bugs, on average.

Contrary to our expectation, we do not observe any decrease in the number of bugs with the introduction of testing policies and increased effort on testing in this project. This motivated us to dig deeper into the state of testing in this company, and to develop hypotheses for further study.

B. Methodology

In *XYZ* every bug, whether identified by a developer or other stakeholders, will be reported in the Jira issue tracker with an issue labeled as a "bug". Each bug report comprises a description of the bug, and the steps that others can follow to reproduce it. Every project in the company is versioned on a Git repository, which is hosted on a Bitbucket server. Developers usually create a new branch once they start to fix a bug, and should mention the issue key when committing a fix. Internally Jira is linked to Bitbucket server, and keeps track of branches and commits that are relevant to an issue. Jira provides a REST API through which we can recover this information.

We could automatically recover commits that fixed 1 119 bugs. Of these bugs, 30% (*i.e.,* 334 bugs) were reported after the testing effort increased in 2017. For each bug we collected commits, file changes associated with those commits, and the developer information.

We were interested to understand the testing practices, and their impact on the quality of the *XYZ* project. In particular, we investigated which components in the project had bugs, why these bugs existed, whether the components were tested before a bug report, and whether developers wrote tests during a bug fix. As it is a non-trivial task to acquire this information for every bug in the project, we randomly chose 200 bugs, and manually inspected their associated commits. Of the 200 bugs, 125 were reported after the introduction of the code coverage policy.

Together with a senior developer in the *ABC* company who was familiar with the *XYZ* project, for every bug, we checked which components are involved. We then inspected whether the code part that caused the bug was covered by a unit test. If so, we analyzed why these tests did not fail and, consequently, did not detect the bug. We manually identified the so-called *"focal methods under test"* to understand the actual purpose of each test case [6]. We also checked whether developers modified, added, or removed tests during the bug fix.

We interviewed the development team members and stakeholders regarding testing practices in the project. We mainly interviewed eight senior developers who each had a minimum of eight years programming experience. The interviews were conducted face-to-face within the company or on the phone, and they were open-ended. In each interview, we usually explained relevant findings in the experiment first, and then asked a question. We encouraged the interviewees to freely discuss any details relevant to the topic. We transcribed the interviews, and analyzed them afterwards.

III. RESULTS

In this section we present the observations that we drew from studying 200 bugs, followed by insights that developers and stakeholders shared with us in this regard.

A. Observations

We analyzed two hundred randomly chosen bugs in depth. Of the 244 components affected by those bugs, only eight components were covered by unit tests. We then evaluated how many bugs affected those eight tested components and the remaining 236 untested ones. Our investigation showed that all but one of the components under test were affected by five or fewer bugs; just one was involved in 19 bugs. Further examinations revealed that this component handles the fingerprinting feature via a new Android API, and it took some time for the developers to learn how to use it in the right way. In contrast to tested components, about a quarter of the untested components suffered from more than five bugs. Furthermore, we could identify eight pairs of recurring bugs, or more precisely similar bugs in the project. We also found a few tests that were commented out during the bug fix,
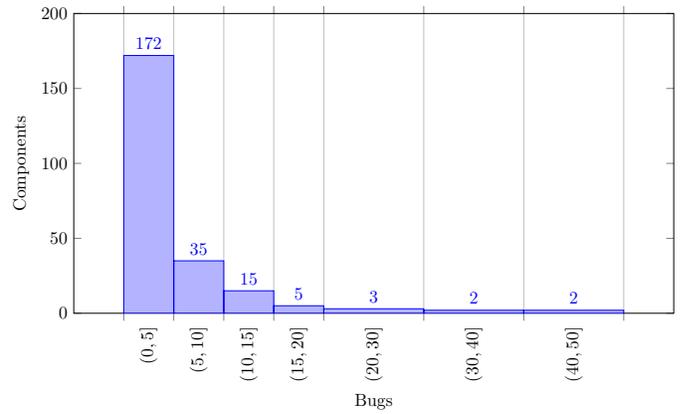

Fig. 2. Bugs per component for 236 untested components

and still commented in the latest version. Figure 2 illustrates the distribution of bugs in the untested components. Tested components appear to be less prone to bugs than untested ones.

For over half of the analyzed bugs, multiple files needed to be changed to fix them. Such cases might be hard to cover with simple unit tests, but might require integration tests. We also discovered that some bugs are due not to coding errors but to changes in the underlying platform or to libraries. Interestingly 20 bugs required no changes to program source file to fix them, but rather to other kinds of files like configuration or resource files that were excluded from the analysis.

We manually inspected the code associated with each of the 200 randomly chosen bugs, and identified 140 cases that we considered "hard to test" based on project domain knowledge. We identified three main reasons: (i) a component violates the single responsibility principle, (ii) it has many dependencies on other parts of the system or external components, or (iii) no exact definitions of correct or incorrect behavior exist, *e.g.,* a progress bar appearing in a wrong location. We observed that developers almost never wrote tests for such cases. In only three cases, *i.e.,* 2%, the defective components were partly covered by tests.

There were only 16 cases that we did not consider hard to test. In seven of those, there were indeed tests that at least partly covered the defective component. Some of those tests existed before the bug was discovered, and some were added when the bug was fixed.

We then asked ourselves what is the quality of the existing tests. We consequently applied mutation testing to assess whether the existing tests could detect changes to the source code that were covered by the tests. Our analysis showed that the existing tests are generally not good at detecting code manipulations, however, tests that were introduced during the release 17.XX onward obtained a significantly higher mutation score in our experiment than older tests. Curiously, none of the new components were heavily modified, though according to the bug evolution (Figure 1) the average number of open bugs increased. To understand the reason for this, we approached the developers.

> *Tested components had fewer bugs than other components. However, a majority of components were hard to test, thus impacting test quality.*

B. Developers' Views

We approached team managers and developers who were mainly involved in the top 15 of the most error-prone components, and interviewed them about their testing practices.

We asked developers *"whether writing tests is important, and if so why the test coverage is so low."* Developers acknowledged the high value of testing, but they also considered it less important (and perhaps less glamorous) than other tasks, such as developing new features. They stated that over roughly a two-year period, the team was under constant pressure to add more features. Furthermore, the communication between the customer and the development team was not very effective. The team had already started implementing new features before all requirements were clear. This led to misunderstandings between the developers and the customers and resulted in many "bug reports", which in fact were changes in requirements. Developers stated that the high frequency of changes demotivated them from writing tests. The team also suffered from high turnover. As a consequence, the project team neglected testing and quality assurance activities, which resulted in many bugs that ended up in production.

The team manager explained that after December 2016, the team decided to improve its Scrum process and follow it more strictly. They improved the quality of communication between the customer and the development team to be more open and better structured. Quality assurance, especially automated unit testing, was given higher priority, and a test coverage goal was introduced and monitored using the SonarQube tool.[1] As a consequence, the quality of the product improved, which is confirmed by feedback of both customers and developers, as well as by a significantly higher mutation score in our experiment. We next queried him regarding the slight increase in the average number of open bugs starting in that period (see Figure 1). He stated that this is a consequence of a "cleanup" measure in which developers (rather than customers) started to systematically report annoying but often non-critical issues, which apparently led to the slight average increase in open bugs. Therefore, the increase in the number of bug reports did not necessarily indicate a decrease in the product quality.

> *Developers were aware of the value of testing, however, frequent changes in the requirements, and pressure to add new features hindered developers from writing tests. Improved communication between stakeholders, and a clear test coverage policy improved the quality of the product.*

In most cases bugs were fixed without making changes to any tests. We asked developers *"when do you write tests?"*. We found that when the effort for developing a unit test for a component appeared to be high, developers tended to abandon writing tests for those components. Developers confirmed that

[1]http://www.sonarqube.org

several components are rather large and play an important role in the application are prone to bugs, but are also very hard to test. For instance, some key components that were hard to test already at the start of the project, grew and became more complex over time. Team members said that they never anticipated the importance of these components in the actual product, and so their testability was not considered at the time. In the experiment, we found three defective components were partly covered by tests, though they were hard to test. Unfortunately, developers of these test cases were no longer available to be asked what motivated them to develop such hard-to-write tests.

> *Developers tended to give up writing tests for a component that is hard to test.*

We asked *"does the discovery of bugs lead to new tests being written"*, and realized that developers do not follow the practice of writing new tests that reproduce reported bugs, though they knew it. We found that developers generally ignore this advice especially when they are confident that they can easily reproduce the bugs manually. They usually fix what they assume to be trivial bugs without writing tests. Of the eight pairs of recurring bugs, all were detected by testers who performed manual tests before rolling out the software. It seemed that none of the developers were aware of recurring bugs, and unanimously agreed that they would have written a test if they had been aware of the recurrence of these bugs. Further discussions with developers revealed that they seldom consider the impact of a bug (*i.e.,* its severity) to decide whether or not to write tests, which is staggering. They mentioned that if a bug is critical, it is fixed immediately and the corrections are deployed as a hotfix. Otherwise, it is moved to the backlog of the project and resolved in one of the next development sprints. We noticed that developers never had a formal training for testing, and tests were only reviewed in an ad hoc way, without the help of specific guidelines.

> *Developers decided to write a test mostly based on their personal opinion rather than upon a clear guideline. The impact of a bug barely influenced this decision.*

Finally, when we asked *"what specific tools do you use to facilitate testing"*, developers only named the SonarQube tool. We then particularly asked if they use any tool, for instance, to (i) recover the traceability links between tests and production code, (ii) assess the quality of tests, and not just their coverage, and (iii) recognize recurring bugs. The answer was mainly negative: few developers knew about a couple of free mutation testing tools, but they said that they crash and do not work as they expected.

C. Threats to Validity

We have drawn our observations based on a single case, which is reflective of other similarly-sized software projects in the *ABC* company. However, the results may not generalize to the state of practice in other companies in Switzerland or abroad.

The findings of this experiment were limited to a subset of randomly chosen bugs in the *XYZ* project that we could automatically link with commits. We manually studied the bugs, and associated commits and test cases, which could have been imprecise due to a lack of experience with the project. We reduced this threat by asking a senior developer in the *ABC* company to help us throughout the whole journey. Also, whenever an issue arose that we could not resolve together, we approached team members and stakeholders of the *XYZ* project.

We did our best to interview the right person for each matter. For instance, to learn about recurring bugs, we asked developers who fixed such bugs. Nevertheless, in cases where the right person was not available we queried a peer in the development team or the manager. Moreover, to reduce bias during the interviews, we kept our questions open-ended, and encouraged the interviewees to discuss any matter related to the topic. The interviews were conducted in German and English, and it may be possible that we misunderstood the answers. We mitigated this threat by double-checking our conclusions with the practitioners.

IV. DISCUSSION

We observed that testing is limited in the *XYZ* project, and from developers' insights we found that the root cause of this phenomenon is that the testability of the project is low.

We see a clear need to encourage developers to start testing early, mainly to ensure testability *i.e.,* the possibility to write automated tests without requiring high effort. We believe realizing the "testability" goal requires commitment from both management and from the development team. For instance, management should establish regular training about software testing, guidelines for writing and reviewing tests, and recognition for testing efforts as being equivalent in value to development tasks. Moreover, developers should commit to a culture of testing. Finally, tool support may exist either as research prototypes or as industrial products, but only few tools are adopted in practice; both industrial applicability and awareness of these tools must be improved.

There are enduring debates in the state-of-the-art work that studies the impact of testing practices. For instance, Bissi *et al.* conduct a systematic review to identify publications that compare the effect of Test Driven Development (TDD) and Test Later Development (TLD) [2]. They conclude from the findings of previous studies that TDD yields more benefits than TLD for internal and external software quality, but it results in lower developer productivity. Pancur *et al.* compare TDD and TLD based on productivity, code coverage, fault-finding capabilities, *etc.* [8]. The experiment is conducted with fourth-year undergrad students during a course on distributed systems. The students are introduced in general to testing as well as TDD and TLD. They conclude that the benefits of test-driven development compared to iterative test-last development are small, and there is no difference regarding productivity. Fucci *et al.* conduct an experiment with professionals in two companies. From four runs of a workshop about unit testing and TDD

at these two companies, they find that the order in which test and production code are written has no important influence on software quality or developer productivity [4]. They conclude that the claimed benefits of TDD may be due to the fact that TDD-like processes encourage fine-grained, steady steps that improve focus and flow. Finally, Borle *et al.* study open source Java projects that have adopted TDD to some extent. They find that this practice is relatively rare in GitHub projects, and that the characteristics of projects with and without TDD are almost the same [3]. These studies were conducted in different and incomparable settings, for example one with a student project, and the other with professionals through a course of workshops. The subjects had different levels of experience, and the effort they put into testing is likely different. Consequently, the level of testability in these projects may vary, and so we cannot easily draw any valid conclusions about the relative benefits of the different practices.

Regardless of the contradictory views on test first or later, we argue the need for *"testability first!"*: we hypothesize that establishment of clear testing policies applied throughout the entire development lifecycle, and early attention to testing of each component are crucial to ensure high testability of the code base, and thus high-quality tests.

In order to draw reliable conclusions about the impact of testing practices, such as whether writing tests before code is advantageous, we therefore see a need to conduct new studies with multiple projects that share the same level of testability: similar effort has been put into testing from management to the development team.

V. RELATED WORK

Kochhar *et al.* interview and survey practitioners to understand the characteristics of good test cases and testing practice [10]. For instance, practitioners agree that high coverage does not mean that a test suite can detect more bugs, and that designing tests that cover different requirements is often superior to maximizing code coverage. They also strongly concur with writing a test that cover a bug fix.

Petric *et al.* study how effectively defective code is actually tested in seven open source Java projects, and show that most of these projects are under-tested. On average about 30% of defective methods are covered by JUnit tests, and the number of methods touched by a JUnit test is strongly related to that test uncovering a defect [11].

Gren and Antinyan study the relationship between unit testing and code quality, and find no correlation between unit test coverage and post-unit test defects [7].

Schwartz *et al.* investigate the cause behind contradictory answers to the question whether coverage criteria have a positive effect on finding faults [12]. They find that test suites with high code coverage are not able to find specific types of faults as frequent as other types. They conclude that code coverage alone does not ensure that faults are triggered and detected, and that the selection of input and oracle can improve the effectiveness of test suites.

Toure *et al.* investigate the ability of a synthetic metric called "Quality Assurance Indicator" to predict early the testing effort in object-oriented software systems [13]. They analyze eight open-source Java projects, and show that the models trained based on this metric have a promising performance to predict the effort required for writing unit test cases.

Garousi *et al.* summarize a pool of 208 papers that help both practitioners and researchers to prepare, measure, and improve software testability [5]. They report that observability and controllability are the two most frequent factors affecting testability, and common ways to improve testability are testability transformation, improving observability, adding assertions, and improving controllability.

VI. Conclusions

We have presented our observations on the state of testing in a large industrial software project. We affirmed that various factors can influence and characterize the testability of a project.

We found that test coverage was relatively low until a deliberate policy was put in place to improve it. Mutation testing revealed that existing tests were often of low quality.

In general developers gave up writing automated tests when the effort was high, though there was incentive to write tests for critical components. When they were pressured to prioritize development of new features over tests, test coverage suffered. A deliberate management policy to improve test coverage led to an increase in tests and in the quality of the code.

Developers were reluctant to test software that undergoes a high rate of change. They typically did not write new tests while fixing bugs, and the bug severity had no impact on writing tests. Developers were not aware of recurring bugs, and therefore such bugs were mostly captured during manual testing.

We observe a lack of attention to the testability of software in the state-of-the-art work that studies the impact of testing practices and methodologies. We hypothesize that establishment of clear testing policies applied throughout the entire development lifecycle, and early attention to testing of each component are crucial to ensure high testability of the code base, and thus high-quality tests. We therefore see a need to conduct new studies with multiple projects that are similar in terms of testability.

Acknowledgment

References

[1] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, March 2019.

[2] Wilson Bissi, Adolfo Gustavo Serra Seca Neto, and Maria Claudia Figueiredo Pereira Emer. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, 74:45 – 54, 2016.

[3] Neil C. Borle, Meysam Feghhi, Eleni Stroulia, Russell Greiner, and Abram Hindle. Analyzing the effects of test driven development in GitHub. *Empirical Software Engineering*, 23(4):1931–1958, Aug 2018.

[4] Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, 43(7):597–614, July 2017.

[5] Vahid Garousi, Michael Felderer, and Feyza Nur Klçaslan. A survey on software testability. *Information and Software Technology*, 108:35 – 64, 2019.

[6] Mohammad Ghafari, Carlo Ghezzi, and Konstantino Rubinov. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70, Sep. 2015.

[7] Lucas Gren and Vard Antinyan. On the relation between unit testing and code quality. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 52–56, Aug 2017.

[8] Matja Panur and Mojca Ciglari. Impact of test-driven development on productivity, code and tests: A controlled experiment. *Information and Software Technology*, 53(6):557 – 573, 2011. Special Section: Best papers from the APSEC.

[9] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six — Mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.

[10] Kochhar Pavneet Singh, Xin Xia, and David Lo. Practitioners' views on good software testing practices. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, 2019.

[11] Jean Petrić, Tracy Hall, and David Bowes. How effectively is defective code actually tested?: An analysis of JUnit tests in seven open source systems. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, pages 42–51, New York, NY, USA, 2018. ACM.

[12] Amanda Schwartz, Daniel Puckett, Ying Meng, and Gregory Gay. Investigating faults missed by test suites achieving high code coverage. *Journal of Systems and Software*, 144:106 – 120, 2018.

[13] Fadel Toure, Mourad Badri, and Luc Lamontagne. Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software. *Innovations in Systems and Software Engineering*, 14(1):15–46, Mar 2018.

[14] Sten Vercammen, Mohammad Ghafari, Serge Demeyer, and Markus Borg. Goal-oriented mutation testing with focal methods. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, pages 23–30, New York, NY, USA, 2018. ACM.