Practices and Tools for Better Software Testing

Spadini, Davide

DOI
10.1145/3236024.3275424

Publication date
2018

Document Version
Accepted author manuscript

Published in
Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conferenceand Symposium on the Foundations of Software Engineering

Practices and Tools for Better Software Testing

Davide Spadini
Delft University of Technology, Software Improvement Group
Delft, The Netherlands
d.spadini@sig.eu
Advisors: Alberto Bacchelli (academic), Magiel Bruntink (industrial)

ABSTRACT

Automated testing (hereafter referred to as just 'testing') has become an essential process for improving the quality of software systems. In fact, testing can help to point out defects and to ensure that production code is robust under many usage conditions. However, writing and maintaining high-quality test code is challenging and frequently considered of secondary importance. Managers, as well as developers, do not treat test code as equally important as production code, and this behaviour could lead to poor test code quality, and in the future to defect-prone production code. The goal of my research is to bring awareness to developers on the effect of poor testing, as well as helping them in writing better test code. To this aim, I am working on 2 different perspectives: (1) studying best practices on software testing, identifying problems and challenges of current approaches, and (2) building new tools that better support the writing of test code, that tackle the issues we discovered with previous studies.

Pre-print: https://doi.org/10.5281/zenodo.1411241

CCS CONCEPTS

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

KEYWORDS

software testing, automated testing, code review, mocking, coupling

ACM Reference Format:
Davide Spadini. 2018. Practices and Tools for Better Software Testing. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3275424

1 INTRODUCTION

Software testing has a fundamental role in any successful software development process [4, 14]. Test cases form the first line of defense against the introduction of software faults and ensure that production code is robust under many usage conditions [4, 6, 20]. Despite this, developers do not test as they should and tend to overestimate their testing effort [2, 15].

In the last decade, a lot of research has been done to understand the negative impact of poor testing on the overall quality of the system. First, van Deursen *et al.* [23] described how the quality of test code was "not as high as the production code [because] test code was not refactored as mercilessly as our production code" [23]. This is in line with the recent studies reporting that developers perceive and treat production code as more important than test code, thus generating quality problems in the tests [2, 3, 25].

In the same work, van Deursen *et al.* introduced the concept of *test smells*, inspired by Fowler *et al.*'s *code smells* [8]. These smells were recurrent problems that van Deursen *et al.* found when refactoring their troublesome tests [13]. Similarly, Palomba *et al.* [16] investigated the relationship between flaky tests and test smells, showing that more than half of flaky tests also contained smells.

Even though research showed that testing is a fundamental process to improve the quality of the project, developers still consider testing less important than "writing features." In my previous work [20], interviewees admitted that developers and managers prioritize production code to tests, because that "is the code running on clients' machines;" tests, on the other hand, are only useful for developers: clients can not know if the new feature they are trying is tested or not. Another discovery of my previous work is that current tools do not fully support developers in writing test code: for example, code review tools are built mainly to support the review of production code [20].

> With my work, I want to (1) raise practitioners awareness of the effect of writing poor tests on the overall software code quality and (2) support developers in writing better test code, by means of new techniques and tools.

To this aim, I will help developers under two different perspectives: first, in better understanding current practices in software testing, and second, building better tools to support them in writing/reviewing test code. Regarding the first point, current practices can help novice developers in understanding how experts handle specific problems when writing and testing software, and they also help expert developers in understanding if their practices are in line with others. On the other hand, when interviewing practitioners I noticed that current software development tools do not have features that may help developers in writing good test code: to fill this gap, I will build new tools that better support the writing/reviewing of tests.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275424

2 TEST QUALITY AND SOFTWARE QUALITY

With the aim of bringing awareness to developers on the effect of poor testing, with these studies I want to investigate the relation between test and production code quality. Since developers consider tests less important than production [20], if we are able to demonstrate that the tests quality has an impact on the production quality, developers may re-consider both as equally important.

2.1 Test Smells to Software Code Quality [22]

In this work we wanted to bring empirical knowledge on the effect of poor testing on the software quality. Van Deursen *et al.* [23] introduced the concept of *test smells*: These smells were recurrent problems that authors found when refactoring their troublesome tests [13]. Given the definition of test smells, we wanted to investigate what is their impact on the overall quality of the system, calculated as change- and defect-proneness of both test methods and the production code they intend to test. To this aim, we conducted a large observational study [5], collecting data from 221 releases pertaining to ten open source software systems, analyze more than a million test cases, and investigate the association between six test smell types and change- and defect-proneness of the code.

Our results show that tests with smells are more change- and defect-prone than tests without smells and production code is more defect-prone when tested by smelly tests. Among the studied test smells, *Indirect testing*, *Eager Test* and *Assertion Roulette* are those associated with highest change-proneness; moreover, the first two are also related to a higher defect-proneness of the exercised production code.

Overall, our results provide empirical evidence that detecting test smells is important to signal underlying software issues. Furthermore, we showed that the presence of design flaws in test code is associated with the defect-proneness of the exercised production code; indeed the production code is 71% more likely to contain defects when tested by smelly tests. This is a call to arms for practitioners, since we finally demonstrated that poor test code is related to poor production code, hence they should be treated as equally important.

2.2 Test coupling: Free your code and tests (in submission at ICSE 2019)

The problem of coupling between test and production code is a well-known problem for practitioners [7, 9, 11]. This problem happens when, due to a refactor or a semantic change in the production code, many tests break. If this is the case, there a chances that tests are tightly coupled to the implementation, namely the tests have too much knowledge of the implementation details of the code. By reducing the coupling between test and production code developers can freely refactor the code in the knowledge that the tests will only fail when they've actually broken something, not just because they applied a different pattern.

In this on-going work, I am studying the problem under different angles: first, we aim at understanding to extent this coupling is a problem by means of quantitative research, namely how spread is the problem? Then, we will look at the cause of the problem and how developers fixed it. Finally, we will discuss with developers different refactorings strategies to tackle the problem in a way that it will never happen again.

If we can detect some refactoring patterns that solve the coupling, we may be able to create a tool that automatically suggests to change/refactor the code according to it.

3 CURRENT PRACTICES IN TESTING

To help developers in writing better test code, we first need to understand how they are currently writing it. To this aim, the works presented in this section are focused on different practices of software testing, with the intention of uncover problem and challenges of current approaches.

3.1 To Mock or Not To Mock? [21]

A widespread practice in software testing is *mocking* [21]. When testing a unit (*e.g.*, a class in object-oriented programming), developers often need to decide whether to test the unit and all its dependencies together (similar to integration testing) or to simulate these dependencies and test that unit in isolation (by means of mocking).

By testing all dependencies together, developers gain realism: The test will more likely reflect the behavior in production [24]; however, some dependencies, such as databases and web services, may slow the execution of the test [12], or be costly to properly setup for testing [19]. By simulating its dependencies, developers gain focus: The test will cover only the specific unit and the expected interactions with its dependencies; moreover, inefficiencies of testing dependencies are mitigated.

We performed a study to *empirically understand how and why developers apply mock objects* in their test suites. To this aim, we analyzed more than 2,000 test dependencies from three OSS projects and one industrial system. We then interviewed developers from these systems to understand why some dependencies were mocked and others were not. We challenged and supported our findings by surveying 105 developers from software testing communities. Finally, we discussed our findings with a main developer from the most used Java mocking framework.

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

This paper has been invited for an extension to the Empirical Software Engineering (EMSE) journal. In the extension, we focused on the evolution of mocks and on the coupling between the mock object and the production code.

3.2 When Testing Meets Code Review [20]

Modern Code Review (MCR) is now a widespread practice in many development projects, since it has been shown that it can improve the quality of the source code [1, 18]. However, past research mainly focused on the review of production code, while there is still a gap

of knowledge on how and why developers review test code. To fill this gap, we conducted a two-phase study to understand how test code is reviewed, to identify current practices and reveal the challenges faced during reviews, and to uncover needs for tools and features that can support the review of test code.

In the first phase, we analyzed more than 300,000 code reviews related to three open source projects (Eclipse, OpenStack and Qt) that employ extensive code review and automated testing. In the second phase, we interviewed developers from these projects and from a variety of other projects (from both open source and industry) to understand how they review test files and the challenges they face.

We found evidence that developers tend to discuss test files significantly less than production files. The main reported cause is that reviewers see testing as a secondary task and they are not aware of the risk of poor testing or bad reviewing. We discovered that when reviewing test files, reviewers often discuss better testing practices, tested and untested paths, and assertions. Regarding defects, often reviewers discuss severe, high-level testing issues, as opposed to results reported in previous work [1], where most of the comments on production code regarded low level concerns.

Among the various review practices on tests, we found two approaches when a review involves test and production code together: some developers prefer to start from tests, others from production. In the first case, developers use tests to determine what the production code should do and whether it does only that, on the other hand when starting from production they want to understand the logic before validating whether its tests cover every path. As for challenges, developers' main problems are: understanding whether the test covers all the paths of the production code, ensuring maintainability and readability of the test code, gaining context for the test under review, and difficulty reviewing large code additions involving test code.

3.3 Test-Driven Review (in submission to ICSE 2019)

My work on how developers review test code [20] has provided initial evidence that some reviewers use the tests to guide their understanding of the code to review. Similarly to how tests can be used to document the corresponding production code, some reviewers have reported to take advantage of tests at review time.

This practice, which we name *Test-Driven Review* (TDR), could be an attractive take on improving the code review process. In fact, tests often have to accompany a change in production code [10, 25], thus adopting this practice would not require more contribution effort, but only a change in how a reviewer would proceed to understand the code under review. The similarities of TDR with scenario-driven inspection, which was effective in the context of code inspections [17], give an additional interesting rationale on why TDR could bring benefits to MCR.

In this paper, we aim at empirically understanding this practice in order to have a more comprehensive view of (i) when and how TDR is applied and (ii) what are its main advantages and disadvantages from the developers point of view. To address these questions and find insights about TDR, we first performed an experiment with 93 developers, analyzing data from more than 150 reviews, and second, we interviewed 9 developers that employ extensive code review and automated testing.

Key findings of our study report that having tests together with production code makes the reviewing of the production faster. Furthermore, we discovered that developers tend to spend more time on the first file it is presented to them, independently whether it is a test or production file, with the consequence of finding more defects on that file.

As for the main implications of this work, we showed that there is a statistically significant difference in the time required to review the production code: Indeed, when it is together with the test code, reviewers take less time to review it compared to when the production code is alone, even though reviewers find the same amount of issues. Hence, reviewers *find the same amount of issues in less time*. Furthermore, as developers tend to more carefully review the first code change they inspect, we believe there is the need for prioritization mechanisms that can effectively rank code changes to be inspected first.

4 BUILDING BETTER TOOLS

While studying best practices in software testing, I identified some weak points of current tools that do not fully support the writing/reviewing of test code. In this section I will explain how I aim to improve software development tools to better handle tests.

4.1 GitHub enhancement

The first tool I want to discuss is the one regarding code review. My previous studies on how developers review test code uncover problems of current MCR tools that do not fully support the review process of test code. I identified 4 main features that may ease the review:

(1) **Linking test and production code:** when reviewing, the files are presented in alphabetical order. This has the disadvantage that all the production files will be presented at the beginning, and all the test files at the end. However, after reviewing a production file, developers would like to immediately see the test file: This results in scrolling the page from the top to the bottom many times. In our tool we aim at adding a link to every production file that will link its test.
(2) **Provide Test-Driven Review (and Production-Drive review)**: similarly, current MCR tools do not provide the possibility to apply TDR. We aim at adding this feature: with the press of a button, all the files will be re-ordered and presented following his/her best practice, namely first the test class and then its production class, or viceversa.
(3) **Detailed code coverage information within the code review:** many IDEs currently provide this information, but it is more difficult to have it within the code review tool. We aim at adding this feature in which it will be possible to see whether the added lines are covered by tests and what is the overall test code coverage.
(4) **Code navigation:** all the interviewees complained about the lack of navigation support in GitHub. For example, it is currently not possible to navigate through the dependencies or go to the definition of a variable/method. This is instead possible in (almost all) IDEs, resulting in reviewers having

to checkout the change and opening it in their IDE. We aim at adding this feature, enabling the reviewer to click on the name of a variable/method and pointing to the definition

The tool is finished and I am currently testing it with practitioners, collecting inputs and feedback in order to improve it.

4.2 Coupling between test and production code

As discussed in Section 2.2, the coupling between test and production code is a well-known problem for practitioners[7, 9, 11]. However, understanding where the problem is and how to fix it is challenging and requires the developers to perform difficult code refactorings.

With our study (Section 2.2) we aim at understanding how spread the problem is and how developers current solve it: the aim is to identify a set of rules that prevent the code to be too coupled. Once completed, we will be able to build a tool that automatically detects classes that are too coupled with the tests, to prevent a cascade of failing tests when refactoring the source code. We aim at doing this within the IDE through static analysis of the code.

Once a pattern is detected, a number of suggestions will appear in the IDE to solve the problem: these suggestions will be a refactoring of the code, for example the creation of a new class, or the refactoring of the test.

5 CONCLUSIONS AND FUTURE WORK

Software testing has a fundamental role in any successful software development process [4, 14]. However, many developers underestimate the effect of poor testing on the overall system code quality, and managers prioritize "new features" to test code. However, in my studies I discovered that test code quality is well related to production code quality, hence they should be treated as equally important.

For this reason, I am working closely with practitioners to understand their best practices when it comes to writing tests, for example studying mocking [21], how they review test code [20], TDR, and test coupling. When investigating best practices in software testing, I uncovered many problems of current tools that do not ease the process of testing. To this aim, I am writing and building new tools that better support developers.

After I finish to build the tools presented in Section 4, my future agenda includes deploying them in a real setting with developers and testers. I plan to do this in SIG [1], the company partner of my PhD. I will be able to collect information on how people use the tools, problems of them and how to improve it. After some iteration, I will have collected enough information to write a research/tool paper and, after that, the tools will be mature enough to be deployed outside SIG.

6 ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 642954.

[1] https://www.sig.eu

REFERENCES

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings - International Conference on Software Engineering*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[2] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman. 2017. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2776152

[3] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 179–190.

[4] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*. IEEE Computer Society, 85–103.

[5] Barry Boehm, Dieter H Rombach, and Marvin V. Zelkowitz. 2005. *Foundations of Empirical Software Engineering*. Springer Berlin Heidelberg. 440 pages. https://doi.org/10.1007/3-540-27662-9

[6] George Candea, Stefan Bucur, and Cristian Zamfir. 2010. Automated software testing as a service. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 155–160.

[7] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2017. Scripted GUI Testing of Android Apps. *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE* (2017), 22–32. https://doi.org/10.1145/3127005.3127008 arXiv:arXiv:1711.03565v1

[8] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. Refactoring: Improving the Design of Existing Code. *Xtemp01* (1999), 1–337. https://doi.org/10.1007/s10071-009-0219-y arXiv:arXiv:gr-qc/9809069v1

[9] Jon Hilton. 2016. Reduce coupling: Free your code and your tests. https://jonhilton.net/2016/03/29/coupling-tests-production/. (2016).

[10] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying fine-grained co-evolution patterns of production and test code. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 195–204.

[11] Robert C. Martin. 2017. Test Contra-variance. http://blog.cleancoder.com/uncle-bob/2017/10/03/TestContravariance.html. (2017).

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

[13] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. 2008. On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. In *Software Evolution*, Tom Mens and Serge Demeyer (Eds.). Springer, 173–202. https://doi.org/10.1007/978-3-540-76440-3_8

[14] Glenford Myers. 2004. *The Art of Software Testing, Second edition*. Vol. 15. 234 pages. https://doi.org/10.1002/stvr.322 arXiv:arXiv:gr-qc/9809069v1

[15] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the diffusion of test smells in automatically generated test code. *Proceedings of the 9th International Workshop on Search-Based Software Testing - SBST '16* (2016), 5–14. https://doi.org/10.1145/2897010.2897016

[16] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *Proceedings of the International Conference on Software Maintenance (ICSME)*. IEEE, 1–12.

[17] Adam A Porter and Lawrence G Votta. 1994. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 103–112.

[18] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. 2008. Open source software peer review practices. *Proceedings of the 13th International Conference on Software Engineering* (2008), 541. https://doi.org/10.1145/1368088.1368162

[19] Hesam Samimi, Rebecca Hicks, Ari Fogel, and Todd Millstein. 2013. Declarative Mocking Categories and Subject Descriptors. (2013), 246–256.

[20] Davide ; Spadini, Maurício ; Aniche, Margaret-Anne Storey, Magiel Bruntink, Alberto Bacchelli, Davide Spadini, and Maurício Aniche. 2018. When Testing Meets Code Review: Why and How Developers Review Tests. 11 (2018). https://doi.org/10.1145/3180155.3180192

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

[22] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On The Relation of Test Smells to Software Code Quality. (2018).

[23] Arie van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*. 92–95.

[24] E.J. Weyuker. 1998. Testing component-based software: a cautionary tale. *IEEE Software* 15, 5 (1998), 54–59. https://doi.org/10.1109/52.714817

[25] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering* 16, 3 (2011), 325–364.