# Evaluation and Improvement of Test Suites

Rainer Niedermayr

CQSE GmbH, Garching b. München, Germany
niedermayr@cqse.eu

## 1 Problem Context

Tests help to ensure software quality and can reveal faults before the software is shipped to customers. Thereby they prevent and reduce failure follow-up costs. Test cases execute parts of the system and compare the observed results with expected ones. They can either be automated or manual.

Automated tests are implemented as executable code or described as scenarios in natural language. An initial effort is necessary to create them and they need to be maintained, e.g. if the specification or data structures of the software change. Apart from that, no human effort is necessary to execute them repeatedly in a software development environment with continuous integration.

Manual tests are carried out by humans. The testers usually have a test specification, perform the described actions and verify the observed results against the expected ones. Some effort is necessary to develop and maintain the test specification. Furthermore, each execution is associated with considerable effort. The advantage of manual tests is that minor changes to the software (eg. to the UI or to internal structures) do not require adjustments.

While most modern systems are tested automatically to a high degree, many systems are still tested manually. Decades-old systems were not designed in a way that they can be tested automatically and this cannot be changed without huge effort and risks. Moreover, some parts of the software – especially graphical user interfaces (GUIs) and hardware interfaces – are difficult to test automatically and would need a high maintenance effort.

## 2 Problem Description

Both automated and manual tests require a lot of effort and thus devour a significant part of the development budget. Although up to 50% of the budget of a typical IT project is spent on testing according to a recent report [4], it is not feasible to fully test a software system. Therefore, it is necessary to focus testing activities on the relevant areas to gain the best cost-benefit ratio. The relevant areas comprise code that is mission-critical or exhibits a high fault-density. In order to cost-effectively improve a test suite, it is promising to augment it by adding test cases for code in the relevant areas that is either not covered or not tested adequately by any of the existing test cases.

## 3 Approach

The idea is to develop measures for evaluating test suites to determine their effectiveness and reveal inadequately tested code, and to propose how to improve the suites.

In a first step, we assessed how thoroughly tested covered code is by applying mutation testing to a set of open-source projects. Methods get executed by test cases and appear as covered, however, this does not necessarily mean that all covered methods were tested in a way

that the tests could have found faults. Therefore, we investigated the expressiveness of code coverage at the method level as indicator for test effectiveness (see Section 4). We want to further analyze methods that were covered but not adequately tested to find indicators that reveal these methods in a static analysis. We plan to investigate metrics such as the minimal distance on the call stack between test case and method, and the number of tests executing a method.

For improving test suites we plan to develop a defect-prediction model that is applicable to real-world systems. Our use case for the model is the augmentation of test suites with additional test cases. We will consider code metrics as well as process metrics to build a model at the cross-project level. In order to narrow down the scope for test developers, we want to propose which methods are fault-prone and need (more) thorough testing, therefore we need to work at the method level. Currently, most existing work in the research area of defect prediction operates at the granularity level of components or files [5]. We plan to validate our approach both with open-source and closed-source systems. Currently, only few studies on the validation of defect-prediction models on commercial systems exist.

We will start by developing an algorithm for classifying changed methods in commits in order to be able to detect bug fixes at the method level. We will integrate the algorithm into the static-analysis tool Teamscale so that we have a framework for analyzing version control systems of software projects and can build up a fault-distribution database. Then, we will combine traditional source code metrics (such as complexity and nesting depth) and process metrics (time span since last change, change frequency in near history) to classify methods that are likely to contain faults. According to previous work ([2], [1]), metrics concerning recent changes are promising indicators. We have access to industrial systems with version control systems and issue trackers and plan to use them when evaluating the defect-prediction model.

Besides the localization of fault-prone code that should be thoroughly tested, we also want to localize code that is not or hardly relevant for testing. First, we plan to carry out an experiment to detect methods that are very trivial and therefore presumably less fault-prone. We expect some methods (e.g. one-line setters and getters, simple delegation methods, very short methods) to relatively unlikely exhibit faults and want to implement a static analysis for their detection. We will evaluate the detected methods empirically using the fault database and developer interviews. Second, since longstanding industrial systems often contain a significant amount of code that is never executed [3] and therefore does not need to be tested, we want to implement a heuristic to detect this code. We assume that code stability and code centrality can be used as predictor: Code that is decentral (i.e. seldom referenced) and stable (unchanged for a long time) might be either perfect or unused. The validation of the concept will be done in a case study in which the computed results will be compared with recorded data of instrumented systems and in developer interviews.

# 4 Results

In order to assess existing test cases, we carried out an experiment to investigate the validity of code coverage as measure for test effectiveness. We implemented and applied a mutation testing approach to investigate code that is covered by test cases. The results of our experiment with 14 open-source projects written in Java show that code coverage at the method level is a valid indicator for the effectiveness of unit tests (in terms of detecting new regression faults) but it is not for system tests that execute large portions of a system [6]. We also found that some of the inadequately tested methods might not be worthy testing due to their triviality or low importance for the software.

# 5 Future Milestones

Q3 2015: Start of the Ph.D.
Q3 2016: Refinement of topic and planning
Q1 2017: Interim presentation
Q1 2019: Thesis complete

# References

[1] Robert M Bell, Thomas J Ostrand, and Elaine J Weyuker. Does measuring code change improve fault prediction? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, page 2. ACM, 2011.

[2] Sebastian Eder, Benedikt Hauptmann, Maximilian Junker, Elmar Juergens, Rudolf Vaas, and Karl-Heinz Prommer. Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice. In *Proc. AST '13*, 2013.

[3] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. How much does unused code matter for maintenance? In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1102–1111. IEEE, 2012.

[4] Worldwide Software Testing Practices Report 2015 - 2016. Technical report, International Software Testing Qualifications Board (ISTQB), 2016.

[5] Jaechang Nam. Survey on software defect prediction. *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[6] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. Will My Tests Tell Me If I Break This Code? In *Proc. CSED '16*. ACM, 2016.