How Good Are My Tests?

David Bowes[1], Tracy Hall[2], Jean Petrić[1,2], Thomas Shippey[1], Burak Turhan[3]

[1]School of Computer Science
University of Hertfordshire
Hatfield, Hertfordshire
AL10 9AB, UK
d.h.bowes, j.petric,
t.shippey@herts.ac.uk

[2]Department of Computer Science
Brunel University London
Uxbridge, Middlesex
UB8 3PH, UK
Tracy.Hall@brunel.ac.uk

[3]M-Group@M3S, ITEE
University of Oulu, POB.3000
Oulu, Finland
burak.turhan@oulu.fi

***Abstract*—Background: Test quality is a prerequisite for achieving production system quality. While the concept of quality is multidimensional, most of the effort in testing context has been channeled towards measuring test effectiveness.**

Objective: While effectiveness of tests is certainly important, we aim to identify a core list of testing principles that also address other quality facets of testing, and to discuss how they can be quantified as indicators of test quality.

Method: We have conducted a two-day workshop with our industry partners to come up with a list of relevant principles and best practices expected to result in high quality tests. We then utilised our academic and industrial training materials together with recommendations in practitioner oriented testing books to refine the list. We surveyed existing literature for potential metrics to quantify identified principles.

Results: We have identified a list of 15 testing principles to capture the essence of testing goals and best practices from quality perspective. Eight principles do not map to existing test smells and we propose metrics for six of those. Further, we have identified additional potential metrics for the seven principles that partially map to test smells.

Conclusion: We provide a core list of testing principles along with a discussion of possible ways to quantify them for assessing goodness of tests. We believe that our work would be useful for practitioners in assessing the quality of their tests from multiple perspectives including but not limited to maintainability, comprehension and simplicity.

***Keywords*-unit testing, test quality, metrics**

I. Introduction

Testing is a paramount activity in ensuring software quality. Automated tests are safety nets when modifying production code [1], however their quality is usually taken for granted or overlooked. Even with the lack of experience and skills, it is possible to fulfil certain criteria, e.g. coverage metrics, without having high quality tests that have the ability to verify and/ or break the system under test (SUT).

There has been serious attempts over the years at identifying the best practices and *test smells* for writing good unit tests, e.g. [1], [2]. We acknowledge that such an objective is subjective by its nature and mostly shaped by experience. For example, a Google search on "unit testing best practices" return many additional "*top-X best practices*" lists (where $X \in \mathbb{N}$) that are likely to vary to a great extent. The critical issue is that applying these best practices is a matter of disciplined choice and manifestation of testing skills. As a result, existing materials serve as a list of recommendations without a (self-)monitoring mechanism.

Assessing whether resulting test artefacts fulfil the goals and follow the principles of testing is quite difficult to assess, because there are different facets of how to define quality within the context of testing. For example, most effort focuses on the "effectiveness" of tests, e.g. coverage, mutation score or the capability to detect errors in general [3], [4], [5], [6], [7], [8], [9]. In their review, Garousi et al. state that among the identified studies only *"(21%) have focused on assessing different quality characteristics of test codes, such as test-code maintainability, understandability, and efficiency."* [10]. However, testing has evolved to a point, particularly in agile development settings, where one should also consider that, e.g. i) tests also act as documentation ii) tests need to be maintained as the system evolves, iii) tests should be simple and expressive to improve readability and comprehension [2], [11], [12]. In other words, looking at test code, one should also be able to identify what the purpose of the test is and how it is implemented, take corrective actions based on the test result, and make modifications when needed.

With respect to the goal of this short paper, we are *not* seeking to compile an exhaustive and authoritative list of principles or best practices. Rather our goal is the early communication of our efforts towards developing a measurement framework to capture the otherwise tacit knowledge about testing principles and supporting practices in the form of expressive metrics that can be used as indicators for "test quality" in an automated manner. This goal is driven by the needs of our industrial collaborator(s) who seek to answer the question:

"How good are our tests?"

In order to practically achieve the above stated goal, we need to limit ourselves to select certain principles among many choices. As stated earlier this is a subjective process. We then investigate whether there is a way to measure them when they manifest in test code, as indicators of test quality. As a result, we have come up with the list of principles that we elaborate in Section II. The list of principles that we discuss in this paper is based on multiple sources selected through the following process:

- As a starting point, we conducted a two-day long semi-structured workshop where we discussed and brainstormed with our industry partners on how to address their need to evaluate the quality of their tests.
- We merged the outcome of the workshop with our teaching experience in delivering introductory and advanced level software testing courses in academia [13] and the industrial tutorial content we have used earlier as part of testing related research in industry [14].
- We checked relevant content from practitioner books whose authors have diverse experience in consulting in software testing [2], [11].
- Finally, we surveyed existing literature (though in a non-systematic way) for proposed metrics to quantify the identified principles and supporting practices, if and when available.

We limit the scope of our work to unit testing. Though the context of our work is limited by the needs of our industry partner, we see no serious impediments in applying our findings to other industrial contexts. In the rest of the paper, we provide the list of selected principles and then discuss related work before we conclude.

II. Unit Testing Principles

In this section, we describe the identified testing principles by providing brief descriptions for each, discussing their practical effects on test code and/or supporting best practices and consider potential ways to infer (in an automated manner) whether a test complies with the given principle.

Principle #1: *Simplicity*
This is the starting and fundamental principle that provides a foundation for all the others. While we have test code in place to check for the correctness of production code, we have few techniques for checking the correctness of test code. As we depend on tests as a safety net, we must rely on their validity and ensure their maintainability through simplicity and consider each test case as small *baby-steps* taken forward. The assumption here is that there is less chance of making mistakes and it is easier to maintain test base when we avoid complexity. In other terms: *"Keep it simple, keep it safe"*.

Compliance with or violation of this principle may be observed in test code in terms of the size of the test code, number of assertions per test case and conditional test logic, all of which can be measured in similar ways to production code [15].

Principle #2: *Readability and Comprehension*
Tests are self-documenting instruments. Hence, the expectation is to have a clear understanding of the intentions of the test code, as with any other document. Besides being simple, expressiveness of tests (including the test case names) is crucial. Using magic numbers, branching, inexpressive naming conventions in test code would disrupt this principle [11]. In addition to checking for constants and branch count in test code, adopting instruments for measuring readability of natural language text, e.g. the Gunning fog index [16], might give indications related to this principle.

Principle #3: *Single Responsibility*
A test should have a single reason to fail, in other words when a test fails one should be able to locate the root cause of the problem. This can be achieved by strictly enforcing verification of one condition per test. Though such a rule comes at the cost of relying on a single data point for the test scenario, triangulation with multiple data points is possible by creating a test case for each. In which case, we must be careful not to duplicate test code and refactor accordingly (e.g. see principle #6). If multiple assertions are allowed or preferred in a single test case, then one should strictly focus on testing one and only one behaviour, which is ideally observable with a single call/entry point to the class under test.

In order to quantify this principle, number of assert statements and number of unique method calls [15] within the assert statements can be used. Calls to multiple methods in class under test might indicate that we are testing more than we should in a single test case.

Principle #4: *Avoid Over-protectiveness*
One common pitfall that violates all preceding principles is the tendency towards over-protectiveness in test code, which is usually visible in terms of redundant assertions in test code [11], for example, checking whether a variable is successfully initialised through *assertNotNull()* as part of tests with other goals. A large number of asserts or multiple types of assert statements appearing together could be related with over-protectiveness.

Principle #5: *Test Behaviour (not implementation)*
Focusing on implementation details would make tests dependent on a particular implementation of specifications. If the preference for implementation details (e.g. choice of underlying sorting algorithm for a given problem) changes through refactoring, implementation focused tests become obsolete and need to be updated towards the new implementation. On the other hand keeping the focus on the expected behaviour and utilising the public interface of the class under test would avoid such problems. Therefore, we recommend focusing on behaviour rather then the implementation details.

Unfortunately, we cannot advise on a straightforward way of capturing this principle, since it is more of a philosophical (or convenience) choice of approach to testing. At this point, however we should make a note on coverage metrics. If one aims at having a certain level of coverage, as many professional environments enforce, it is relatively easy to reach high coverage without testing expected behaviour. On the other hand, addressing expected behaviour would indirectly yield a high level of coverage. Therefore, interpretation of coverage metrics, as a sign of test quality, should be avoided, as existing research also questions its relation with test effectiveness, e.g. [17], [18], [19]. The bottom line is that coverage should not be the explicit goal, yet it will be achieved with a focus on testing behaviour.

Principle #6: *Maintainability: Refactoring test code*

Tests are helpful for easy maintenance of code, yet tests should be easily maintainable as well. As the production code evolves, it is inevitable that the test code will evolve with it. Whether this will cause an additional burden depends on how maintainable the test code is. Maintainability for test code is a goal as well as it is a principle to keep in mind. Simplicity and refactoring of test code are key supporting principles in this aspect.

Test code duplication should be avoided. Based on our experience, duplication of test code is a common practice to have a fast start when writing a new test case. It would make maintainability harder in the long run, unless test code is refactored [20]. We should also keep in mind that test code is also *code*. Checking for the amount of test code duplications [15] and number of refactorings applied (e.g. checking the evolution of test code in terms of complexity over time [18] or recording the number of IDE supported refactoring actions when possible) would be possible ways of capturing this principle.

Principle #7: *A test should fail*
While this seems pretty straightforward, it is one of the common mistakes we observe at least among students who are future professionals. A never failing test is not helpful at all, and a test can simply achieve such a god-mode if one does not insert any assertions in the test case. Testing frameworks do not (but should) recognise such simple mistakes. As a principle one should see a test's ability to fail, which is one of the driving arguments of test-driven development [21]. In addition, it is a good practice to *fail()* incomplete tests for the same reason. Ensuring that there is at least one assert statement [15], other than *assertTrue(true)*, in a test case would partially capture this principle. Having a history of test runs can also give an idea whether a test has ever failed or not.

Principle #8: *Reliability*
In order to rely on tests as a safety net for development, they should behave as expected, that is they should consistently pass or fail under fixed conditions. The result of a test cannot involve randomness and if there is any source of non-determinism, these should be isolated and removed. This is one of the principles that is usually hard to capture and taken for granted.

Principle #9: *Happy vs. Sad tests*
This principle is associated with the goals of testing: to verify the system (also known as happy tests) vs. to break the system (also known as sad tests). Due to the phenomenon known as confirmation bias, it does not come as a surprise to see testers' tendency to confirm behaviour rather than to break it [22]. While it might be useful to explicitly state that the goal of testing is to discover faults [23], we believe that both views on the goals of testing have merits and should be considered simultaneously during the testing process. Considering testing strategies such as partitioning strategy and boundary case analysis, we would expect at least as many sad test cases as happy test cases. In this respect, Causevic et al. propose making sad test case construction an additional step in test-driven development [24], [25]. Unfortunately, it is not possible to automate the measurement of this principle without content-based analysis. A good practice would be to indicate the type of the test as a part of the test case name as a standard naming convention, which would allow for an automated analysis.

Principle #10: *Tests should not dictate the code*
This principle, to which Meszaros refers to as *Keep Test Logic Out of Production Code* principle [2], suggests that we should not make any modifications in production code for the purpose of testing. In essence, testing logic should not propagate into production code. A common example is to include a production code method that is only called from within the test code to access internal states of the class under test. Another example is to update access modifiers of production code methods, e.g. from private to public, to be able to reach them from the test code. Both examples are also related to principle #5, testing behaviour vs. implementation, and both cases can be avoided with a behaviour focused testing approach. One way to detect such cases is to investigate the fan-in (or static call graph) for production code methods and to check whether the incoming calls are from test code only. Tests should also avoid the use of reflection to access private methods and fields as this also relies on a knowledge of implementation which may not be matched by a change in behaviour.

Principle #11: *Fast feedback*
One of the goals of test automation is to provide timely feedback regarding the results of the tests. No developer would prefer to wait for the completion of unit test runs before making progress. Delays in test runs can be due to logical flaws in the implementation of the test or production code that introduce unwanted complexity or memory leaks. Alternatively, the class under test might be relying on external resources that take a long time to respond. In either case, long test run times are indicators of problems and should be investigated further. Clearly, the metric to monitor this principle is the run-time for test cases.

Principle #12: *4-phase test design*
This principle is yet another simple but commonly overlooked one especially by novice developers. The 4-phase test design suggest the setup, execution, validation and tear-down stages, in the given order, in constructing test cases. Our experience with students shows that it is surprisingly common to skip (forget) the setup or the execution stages. This can be checked by investigating the number of non-assert lines before the first assertion in a test case, i.e. we would expect at least one line to be executed even if all setup is handled by a shared fixture. However, we should also note that there might be cases where the execution is embedded in the fixture. In terms of expressiveness of tests, it is good practice to refactor the fixture and include the execution stage within the test in such cases.

Principle #13: *Simplicity of fixtures*
As the number of test cases grow in a test class, a certain pattern emerges where the code needed for the setup phase (of the 4-phases discussed above) is duplicated. In order to

avoid code duplication, we refactor this shared code fragment into a special test method called the fixture. However, as we keep adding more test cases, the fixture is updated and it tends to include code fragments used only by a subset of test cases that are not needed by the other test cases which share the same fixture. As a result, the fixture gets longer, more complex and harder to maintain. That's why we should keep this specific principle in mind to ensure the other principles we have discussed are in place for the simplicity of the fixture; that is, fixtures should be minimal. Size and complexity of the code in fixtures can be used as proxies to control for this dedicated principle [26].

Principle #14: *Test (in)dependency*
We should be able to run our tests in any order and in isolation and tests should not rely on each other in anyway. This would enable taking baby-steps and allows us to add new test cases without considering any dependencies or any effects they might have on existing test cases. Indeed, popular testing frameworks do not guarantee the order the tests are run and provide us with the capability to run a selection of existing tests. This principle would be clearly violated if a test invokes another test or if there are static fields in class under test that would preserve their latest state even when testing framework creates a new instance of the class. Checking for fan-in to test case methods (one would expect zero for all cases) and for static fields in production code would be helpful for assessing this principle.

Principle #15: *Use of test doubles*
When we are testing for classes that rely on external resources that do not yet exist or might introduce delays in the run time for the completion of tests, we should make use of test doubles in order to stub/mock out external dependencies for the testability of target class. Since the use of test doubles are case specific and context dependent, it is not possible to make a generic recommendation. Nevertheless, having the information whether test doubles are used, e.g. *isMocked()*, would be an indicator to check for this principle.

III. Related Work

In this section, we briefly discuss existing work related to our study.

Garousi et al. report in their systematic literature review on test-code engineering that one of the two main categories of test quality assessment research focus on detection of *test smells* covering 61% of the identified studies.

Test smells are defined by van Deursen et al. in their seminal paper "Refactoring Test Code" [1]. They introduced the concept of test smells and identified 11 of them: mystery guest, resource optimisation, test run war, general fixture, eager test, lazy test, assertion roulette, indirect testing, for testers only, sensitive equality and test code duplication.

Empirical evaluation of the validity of test smells have been demonstrated by various studies. For example, van Rompaey et al. quantify test smells and evaluate them with open source case studies [26], [27], and Bavota et al. provide evidence on the existence of test smells in open source and industrial systems and their negative impact on comprehension and maintenance [28], [29].

Breugelmans and van Rompaey developed a tool (TestQ) in order to quantify the test smells and guide testers through measurement and visualisation [15]. Greiler et al. also developed and evaluated a tool (TestHound) in three industrial case studies focusing on fixture related smells that makes recommendations for refactoring [30].

A summary of the 15 principles identified in this work and their comparison with the test smells are provided in Table I. The last column of Table I maps the principles to relevant test smells when applicable, whereas the third column (**Metric(s)**) list the proposed metrics for quantification of the principle. In the third column, we also indicate whether TestQ tool provides support for automated analysis, e.g. with *(TQ)* label next to the metric's name. We have identified eight principles that do not map to existing test smells (this is expected as our intentions are different than of van Deursen et al.) and propose metrics for six of them. We have identified additional potential metrics for the seven principles that partially map to test smells.

To the best of our knowledge, Nagappan's STREW Metric Suite is one of the first attempts to explicitly capture the quality of a test suite [31], [32], [33]. Their metric suite is based on metrics derived from number of assertions and CK metrics in relation to the size of test suites. As for using assertions as a proxy for test quality, Aniche et al. reports that higher number of asserts are indicators of problems in production code, based on their analysis of open source and industrial projects [34]. Similarly, Vahabzadeh et al. analyse the bug content of test code in open source projects and indicate that *"...incorrect and missing assertions are the dominant root cause of silent horror bugs... the majority of false alarm bugs happen in the exercise portion of the tests"* [35].

Herzig utilises test execution metrics successfully, though with a different objective, which is to predict post-release production defects [36].

Lanubile and Mallardo observe improvements on the quality of test code after code inspections [37]. Daka et al. emphasise the importance of the readability of tests and propose a domain specific model for test readability [38]. Though neither approach is feasible for automation, these studies address the notion of test quality.

IV. Conclusions and Future Work

In this paper we aimed to identify a core list of testing principles that address multiple facets of quality and discussed how they can be quantified as indicators of test quality. We have conducted a two-day workshop with our industry partners to come up with a list and then utilised our academic and industrial training background and the recommendations in practitioner oriented testing books to refine the list. We surveyed existing literature for potential metrics to quantify identified principles.

We have identified 15 principles in total where eight principles do not map to existing test smells. While we cannot

TABLE I
SUMMARY OF PRINCIPLES AND THEIR POSSIBLE QUANTIFICATION IN RELATION WITH TEST SMELLS DEFINED IN [1].

Principle #	Principle Name	Metric(s)	Relevant Test Smell
1	Simplicity	Size *(TQ)* Complexity Branch Count *(TQ)* #assertions	IndentedTest VerboseTest
2	Readability and Comprehension	#constants Branch Count *(TQ)* Gunning fog index	IndentedTest
3	Single Responsibility	#assertions #uniqueMethodCallsInAssertions *(TQ)*	EagerTest
4	Avoid Over-protectiveness	#assertions #assertionType	
5	Test behaviour (not implementation)	n/a	
6	Maintainability: Refactoring test code	duplicate detection *(TQ)* #refactoring Complexity over a period	DuplicatedCode
7	A test should fail	#assertions > 0 *(TQ)* #assertTrue(true) test run history	Assertionless EmptyTest
8	Reliability	n/a	
9	Happy vs. Sad tests	n/a (unless indicated with naming convention)	
10	Test should not dictate code	fan-in production code (from test code) *(TQ)*	ForTestersOnly
11	Fast feedback	runtime for tests	
12	4-phase test design	non assert statements before first assertion	
13	Simplicity of fixtures	Size *(TQ)* Complexity Branch Count #assertions$= 0$	GeneralFixture
14	Test (in)dependency	fan-in test code ($= 0$) #staticFields	
15	Use of test doubles	isMocked()	

quantify two of these additional eight principles, we were able to propose metrics for the remaining six. Further, we have identified additional potential metrics for seven of the principles that partially map to test smells.

Our contribution with this work is to provide a core list of testing principles focusing on different quality aspects other than effectiveness or coverage, along with a discussion of possible ways to quantify them for assessing goodness of tests.

It should be noted that we report work-in-progress in this short paper and our future intent is to extend our work by refining the list as necessary, conducting a systematic review of existing literature and providing tool support to enable automated monitoring of test quality.

ACKNOWLEDGMENTS

This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number EP/L011751/1 and by Academy of Finland with decision number 278354. Authors would like to thank Liam Williams and Dean Smith for their invaluable opinions that contributed to this work.

REFERENCES

[1] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," 2001.
[2] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.
[3] Y. Chernak, "Validating and Improving Test-Case Effectiveness." *IEEE SOFTWARE*, vol. 18, no. 1, pp. 81–86, 2001.
[4] D. Delgado and A. Martinez, "Cost Effectiveness of Unit Testing: A Case Study in a Financial Institution," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, pp. 340–347.
[5] M. Ellims, J. Bridges, and D. C. Ince, "The Economics of Unit Testing," *Empirical Software Engineering*, vol. 11, no. 1, pp. 5–31, Feb. 2006.
[6] R. Gopinath, C. Jensen, and A. Groce, *Code coverage for suite evaluation by developers*. New York, New York, USA: ACM, May 2014.
[7] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*. IEEE, 2009, pp. 291–301.
[8] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Programs, tests, and oracles: the foundations of testing revisited." *ICSE*, pp. 391–400, 2011.
[9] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, Dec. 1997.
[10] V. Garousi Yusifoğlu, Y. Amannejad, and A. Betin Can, "Software test-code engineering: A systematic mapping," *Information and Software Technology*, vol. 58, pp. 123–147, Feb. 2015.
[11] L. Koskela, *Effective Unit Testing: A guide for Java developers*. Manning Publicationg, 2013.
[12] R. Osherove, *The Art of Unit Testing With Examples in .NET*. Manning Publicationg, 2009.
[13] D. Fucci and B. Turhan, "On the role of tests in test-driven development: a differentiated and partial replication," *Empirical Software Engineering*, vol. 19, no. 2, pp. 277–302, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9259-7
[14] D. Fucci, B. Turhan, N. Juristo, O. Dieste, A. Tosun-Misirli, and M. Oivo, "Towards an operationalization of test-driven development skills: An industrial empirical study," *Information and Software Technology*, vol. 68, pp. 82 – 97, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584915001469
[15] M. Breugelmans and B. V. Rompaey, "Testq: Exploring structural and maintenance characteristics of unit test suites," in *International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)*, Paphos, July 2008.

[16] R. Gunning, “The fog index after twenty years,” *Journal of Business Communication*, vol. 6, no. 2, pp. 3–13, 1969.

[17] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness.” *ICSE*, pp. 435–445, 2014.

[18] D. Tengeri, Á. Beszédes, T. Gergely, L. Vidács, D. Havas, and T. Gyimóthy, “Beyond code coverage - An approach for test suite assessment and improvement.” *ICST Workshops*, pp. 1–7, 2015.

[19] Y. Wei, B. Meyer, and M. Oriol, “Is Branch Coverage a Good Measure of Testing Effectiveness?” in *Empirical Software Engineering and Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 194–212.

[20] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, “On the interplay between software testing and evolution and its effect on program comprehension,” *Software evolution*, 2008.

[21] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[22] G. Calikli and A. Bener, “Empirical analysis of factors affecting confirmation bias levels of software engineers.” *Software Quality Journal*, vol. 23, no. 4, pp. 695–722, 2015. [Online]. Available: http://dblp.uni-trier.de/db/journals/sqj/sqj23.html#CalikliB15

[23] B. Meyer, “Seven Principles of Software Testing.” *COMPUTER*, vol. 41, no. 8, pp. 99–101, 2008.

[24] A. Causevic, S. Punnekkat, and D. Sundmark, “Quality of testing in test driven development.” in *QUATIC*, J. P. Faria, A. R. da Silva, and R. J. Machado, Eds. IEEE Computer Society, 2012, pp. 266–271. [Online]. Available: http://dblp.uni-trier.de/db/conf/quatic/quatic2012.html#CausevicPS12

[25] A. Causevic, D. Sundmark, and S. Punnekkat, “Test case quality in test driven development: A study design and a pilot experiment.” *EASE*, pp. 223–227, 2012.

[26] B. Van Rompaey, B. Du Bois, and S. Demeyer, “Characterizing the Relative Significance of a Test Smell.” *ICSM*, pp. 391–400, 2006.

[27] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, “On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 12, pp. 800–817.

[28] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, “An empirical analysis of the distribution of unit test smells and their impact on software maintenance.” *ICSM*, pp. 56–65, 2012.

[29] ——, “Are test smells really harmful? An empirical study,” *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, May 2014.

[30] M. Greiler, A. van Deursen, and M.-A. Storey, “Automated Detection of Test Fixture Strategies and Smells,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 322–331.

[31] N. Nagappan, “Toward a software testing and reliability early warning metric suite,” in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE, 2004, pp. 60–62.

[32] N. Nagappan, M. A. Vouk, and J. A. Osborne, “Early estimation of software quality using in-process testing metrics: a controlled case study.” *ACM SIGSOFT Software Engineering Notes ()*, vol. 30, no. 4, pp. 1–7, 2005.

[33] N. Nagappan, L. Williams, J. A. Osborne, and M. A. Vouk, “Providing Test Quality Feedback Using Static Source Code and Automatic Test Suite Metrics.” *ISSRE*, pp. 85–94, 2005.

[34] M. F. Aniche, G. A. Oliva, and M. A. Gerosa, “What Do the Asserts in a Unit Test Tell Us about Code Quality? A Study on Open Source and Industrial Projects.” *CSMR*, pp. 111–120, 2013.

[35] A. Vahabzadeh, A. M. Fard, and A. Mesbah, “An empirical study of bugs in test code,” *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 101–110, 2015.

[36] K. Herzig, “Using Pre-Release Test Failures to Build Early Post-Release Defect Prediction Models.” *ISSRE*, pp. 300–311, 2014.

[37] F. Lanubile and T. Mallardo, “Inspecting automated test code: a preliminary study,” *Agile Processes in Software Engineering and ...*, 2007.

[38] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, “Modeling readability to improve unit tests,” in *the 2015 10th Joint Meeting*. New York, New York, USA: ACM Press, 2015, pp. 107–118.