A New Dimension of Test Quality: Assessing and Generating Higher Quality Unit Test Cases

Giovanni Grano
Software Evolution and Architecture Lab, University of Zurich
Zurich, Switzerland
grano@ifi.uzh.ch

ABSTRACT

Unit tests form the first defensive line against the introduction of bugs in software systems. Therefore, their quality is of a paramount importance to produce robust and reliable software. To assess test quality, many organizations relies on metrics like code and mutation coverage. However, they are not always optimal to fulfill such a purpose. In my research, I want to make mutation testing scalable by devising a lightweight approach to estimate test effectiveness. Moreover, I plan to introduce a new metric measuring test focus—as a proxy for the effort needed by developers to understand and maintain a test— that both complements code coverage to assess test quality and can be used to drive automated test case generation of higher quality tests.

CCS CONCEPTS

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

KEYWORDS

Software Testing, Test Quality, Automated Testing

ACM Reference Format:
Giovanni Grano. 2019. A New Dimension of Test Quality: Assessing and Generating Higher Quality Unit Test Cases. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3293882.3338984

1 PROBLEM STATEMENT

Our society runs on software: every sector—included the most critical ones—relies on software at different level. Therefore, serious software bugs might have a huge impact on the society itself. Software testing is the set of activities that aims at preventing the introduction of such bugs with unit testing acting as the first defensive line in preventing that. For instance, in a typical continuous integration (CI) pipeline, unit tests run as soon as changes to the code base are performed with the goal of assessing their quality. However, not all the unit tests are the same: some of them might suffer of a poor overall quality. To assess test quality, most organizations mainly relies on metrics like code or mutation coverage. Unfortunately, the former has been shown to be a non-optimal indicator when it comes to measure test quality [16]. The latter is used as strongest alternative to code coverage [17] to estimate test *effectiveness*, *i.e.*, the ability of revealing bugs. However, mutation testing has the main disadvantage of being extremely computationally expensive [17]: thus, especially in case of large software systems or when the frequency of commits is high (*e.g.*, in CI), systematic mutation testing becomes hard to put in practice. Multiple approaches have been proposed to address such a scalability problem [22]. However, Gopinath *et al.* [10] showed that most of them do not provide a practical gain if compared to a random selection of mutants. Therefore, the problem of assessing test effectiveness on a large scale is still far from being solved.

Effectiveness is not the only desirable quality that tests should have. As well as source code, test cases need maintenance since they evolve over time along with the code they exercise [21, 30]. In some cases, a new feature might change the behavior of the code and make the tests obsolete; on the other hand, changes might introduce bugs that are then revealed by tests. Therefore, developers need respectively either modify the tests or fix the production code [30]. In both cases, understanding the test is the first step prior any kind of maintenance task. Test automation expert Meszaros described in his milestone book 68 patterns for making tests easier to understand and maintain [20]. One of the most important, the *Single-Condition Tests* principle [20], suggests that maintainable tests should not verify many functionalities at ones to avoid test obfuscation. I refer to this property as *test focus*, with the underlying hypothesis that focused tests are easier to understand and therefore, they ease fault localization and debugging tasks.

If test quality assessment is still mainly measured by code and mutation coverage, the same is true for automated test case generation [31]: despite decades of research on the topic, code coverage is still used as the main criteria for the evolutionary search [31]. This results in test suites that reach high coverage but suffer of low overall quality [23]. Many authors recently investigated non-coverage—quality—aspects in addition to the high code coverage [18]. However, the problem of balancing two contrasting objectives without detrimental effects in one of the two remains far from being solved, especially in a multi-target fashion [7].

2 PROPOSED RESEARCH

In my research, I propose to address the scalability problem of mutation testing estimating test effectiveness by using source-code-quality indicators. Moreover, I want to complement the usage of code coverage metrics to evaluate test quality—as well as to drive

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338984

test case generation—by introducing a new *focus* metric that measures for the effort needed to understand test code.

The underlying hypothesis of my research is the following:

> *Metrics that measure the effectiveness and the focus of a test can be used to assess and improve the quality of test suites. Relying on unsupervised methods, we can pinpoint the existence of low quality tests that should be improved. Such metrics can be used to automatically generate high-quality and effective tests.*

In the following, I motivate the research questions that are used to validate my hypothesis and discuss how they contribute to support the specification, verification, and execution of experiments.

At first, I want to understand more in details which are the factors that make a test *effective* or *non-effective*. This would allow to recommend developers which source-code aspects to look at with the goal of increasing the overall test-suite effectiveness. Secondly, I aim at designing new lightweight approaches that aims at predicting the test-effectiveness without the need to actual go through the mutation testing pipeline. This would allow tackle the main disadvantage of mutation testing, *i.e.*, its high computational cost.

RQ 1 *To what extent can we estimate test-case effectiveness using source-code-quality indicators?*

While the ability to reveal faults remains the main goal of tests, it is important that they ensure other qualities. Amongst the others, I want to concentrate on test understandability that is crucial for different reasons. In particular, understanding the test is the first step for developers in case (i) a test fails and the fault needs to be located, or (ii) the production code changes and the related tests need to be modified accordingly. To estimate the effort spent by developers in understanding a test, I plan to design a new metric based on the concept of entropy [15] with the explicit goal to capture the *focus* of a test, measuring to what extent it complies to the *Single-Condition Tests* principle [20]. My hypothesis is the following: less entropic (*i.e.*, more focused) tests are easier to understand and thus, they ease debugging tasks for developers.

RQ 2 *To what extent can we measure the focus of a test?*

Finally, in the last research question I plan to design an adaptive framework for automated test case generation able to optimize any non-coverage aspect without a detrimental effect on the code coverage. This would allow to put quality aspects into the loop of automated generation, thus, resulting in higher quality tests.

RQ 3 *Can we improve the quality of automatically generated test-cases?*

Each of those research questions are related to the formulated hypothesis. In particular, $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ tackles two different dimensions of test quality, respectively *effectiveness* and *focus*, as a proxy for understandability. The third research question aims at prosing a general framework for test case generation that allow to consider any other criteria along with code coverage.

3 RQ1: ABOUT TEST-CASE EFFECTIVENESS

To answer $\mathbf{RQ}_1$, I conducted a study [11] with a twofold goal: understanding the factors that discern effective from non-effective tests; predicting test-case effectiveness —as indicated via mutation testing— in a lightweight fashion relying on the identified factors.

3.1 Factors Impacting Test-Case Effectiveness

We conducted a study [11] to understand the differences in the distribution of different factors for test cases having *high* or *low* mutation scores. In particular, we considered 67 different factors extracted from both production and test code, coming from five different dimensions, *i.e.*, Code Coverage, Test Smells, Code Metrics, Code Smells and Readability. To assign test cases to either the *effective* (*i.e.*, high score) or the *non-effective* (*i.e.*, low score) set, we firstly conducted a mutation analysis; then, we used the quartile of the obtained mutation score: the first quartile denoted the *non-effective* set, while the fourth quartile the *effective* one. Therefore, to compare the distribution of each factor in the two sets of tests we apply the Wilcoxon Rank Sum statistical tests [6]. Moreover, we estimate the *magnitude* of the observed differences using the Cliff's Delta, a non-parametric effect size measure for ordinal data [14].

We discovered that effective tests statistically differ from non-effective ones for 41 out of the 67 (*i.e.*, about 61%) investigated factors. This result suggests that source-code quality metrics might be relevant for assessing test-case effectiveness. Our main findings were that a test case is more effective when it has a high statement coverage and does not contain test smells. Moreover, 20 metrics computed on the production classes had a relevant impact on effectiveness: this confirmed the common wisdom that the larger and complex the production code is, harder is for the tests to reveal faults. Similarly, we observed that higher quality and complexity of the test code translates in higher ability to find faults. The same seemed to be true looking and the relation between code-smells and effectiveness, with two investigated factors having a statistically significant negative relationship with the mutation score. To sum up the findings of our study, we observed that a test case tends to be more effective when it has a high statement coverage and does not contain test smells. Moreover, the absence of design flaws in the CUTs and its quality represent strong factors for test effectiveness.

3.2 Predicting Test-Effectiveness

Relying on the same source-code-metrics we exploited to understand the relation between such factors and the effectiveness, we devised a technique able to predict effectiveness itself, as indicated by mutation analysis [11]. Such a technique is orthogonal to existing approaches and aims at predict effectiveness by using source-code-quality indicators computed on both production and test code (e.g., quality metrics [5] or code/test smells [35]). Therefore, such approach resulted to be extremely lightweight being based on source-code metrics that can be easily calculated or even coming from free in Continuous Integration (CI) environments. We devised and evaluated two different prediction models for test-case effectiveness: the first model (referred as *dynamic*) exploits all the 67 aforementioned factors; the latter (referred as *static*), on the contrary, uses the same factors but the statement coverage, which is the only dynamic metrics—*i.e.*, that requires code execution to be computed—we considered. Before being able to properly evaluate the prediction models, we went through a series of preprocessing steps. At first, we applied data normalization (or features scaling). Then, we applied a feature selection algorithm to select the possible subset of features in order to identify the ones giving the best performance. Finally, we applied a grid search method to tune the parameters of the different

Table 1: Performance of the RFC on nested cross-validation.

	Acc.	Prec.	Rec.	F1	AUC	MAE	Brier
Dynamic	0.948	0.940	0.960	0.949	0.949	0.051	0.035
Static	0.864	0.864	0.865	0.864	0.864	0.137	0.095

employed algorithms: in total, we experimented the performance of a Random Forest, a K-Neighbors and a Support Vector Machines classifiers. To train, evaluate and select the best algorithms, we adopted a 10-fold nested cross-validation approach. Table 1 reports the results of such evaluations for the best model, *i.e.*, the Random Forest Classifier (RFC). We demonstrated that prediction models can be effectively exploited to classify test-case effectiveness. A model relying on both dynamic and static information achieves performance close to 95% in terms of F-Measure and AUC- ROC, while the performance of a model only using static indicators decreases of about 9% only: we argue that this model still gives good performance and is moreover a more practical solution in a real scenario. Indeed, most of the metrics that we consider in the static model are already computed by software analytics tools employed in Continuous Integration (CI) pipelines.

4 RQ2: ENTROPY TO CAPTURE THE FOCUS

Test effectiveness is not the only desirable quality for test cases. Other qualities like readability and understability are crucial, especially considering that tests are not static artifacts but they evolve along with the application they exercise [30]. In a preliminary work [12], we showed that readability of tests is lower compared to the production code they exercise. In the context of this second research question I want to focus on test understandability. Understanding test behavior is the first crucial step both in case (i) the production code changes and some tests need to be updated, and (ii) a test fails and the source code needs to be fixed [30]. Estimate the degree of understandability and maintainability of a test is very hard in practice. I plan to address such an issue by developing a new metric based on the Information Theory concept of entropy [33] that has already been used in previous SE research [15]. In particular, I plan to calculate the entropy of the coverage over the production code for a given test. This formulation aims at measuring to what extent a test complies to the *Single-Condition Tests* principle [20]: I refer to this property as test *focus*. According to the definition of entropy, if a test covers branches belonging to a single method, the entropy of its coverage is 0, *i.e.*, the test is highly focused. On the opposite case, if many methods are called and the test covers the exact same number of branches for all of them, the entropy is maximum and the test is not focused.

I plan to conduct both a qualitative and a quantitative analysis involving a large number of controlled experiments with actual software developers structured as follows. I will select a set of Java classes and I will manually inject a bug in a precise position. Then, I will either write or generate two test cases able to reveal the introduced bug, respectively with high and low values for the entropic metric. The experiment will be composed of many tasks, each of them consisting in locating the manually injected bug inspecting both the Java class and a provided test, either with high or low entropy. For each task I will collect: (i) if the participants correctly locate the bug and (ii) the time they need to do it. Moreover, after each task I will ask the participants to rank the understandability of the received test in a scale between 1 and 5. I aim to verify the following hypothesis: *developers are faster in locating bugs if focused tests are provided.* This would eventually confirm that the entropic metric is well suited to measure test understandability.

5 RQ3: ADAPTIVE TEST-CASE GENERATION

The main promise of automated test case generation is to reduce the burden of developers in manually writing test cases. However, despite decades of research on this topic [19], automatically tests generation is still far to be used in practice. One of the main reasons lie in the fact that the resulting tests suffer of low overall quality [23]. One possible solution is to include quality aspects as objectives to be optimized during the process of test case generation. While most of the research effort so far has been devoted to maximize code coverage criteria [19], recently research attempted to put *non-coverage* aspects into the loop [7, 18, 24]. Such works share the same formulation, *i.e.*, they consider coverage and non-coverage criteria as equally important for the evolutionary search. However, it has been empirically shown that balancing two contrasting objectives often results in a detrimental effect on the achieved coverage [7]. Moreover, these studies propose a single-target approach rather than a multi-target one, while the latter has been shown to be more effective, addressing the main issues of the former [29]. Tackle this issue is the goal of my $\mathbf{RQ}_3$. To achieve this, during the evolutionary search I plan to dynamically detect stagnation in the first objective, *i.e.*, code coverage. In particular, stagnation is detected when no improvement in the fitness function is observed for the uncovered targets. In such a case, the algorithm will temporarily disable the optimization for the secondary objective.

I plan to implement such an adaptive approach in EvoSuite [8] by extending the base algorithm of MOSA [27]. Thus, I plan to empirically evaluate the adaptive algorithm as follows. I will select a secondary objective to maximize along with code coverage: possible objectives might be execution time or memory consumption. Then, I will select a large set of Java classes from the SF110 benchmark [28] and I will compare the test suites generated over such classes by the baseline—*i.e.*, MOSA—and the proposed adaptive approach. Such a comparison will involve not only the chosen secondary objective, but also branch coverage and mutation score to double-check whether the optimization of the non-coverage objective results in a detrimental effect on either code coverage or fault effectiveness. To perform this analysis, I plan to use the non-parametric Wilcoxon Rank Sum Test [6] and the Vargha-Delaney [37] test to measure the magnitude of the differences.

6 RELATED WORK

Test Effectiveness. Research mainly focused on code coverage and mutation testing to evaluate test effectiveness. The role of code coverage in fault localization [39, 40] and detection [4] has been previously explored. Despite some encouraging results, Rojas and Fraser [31] stated that the coverage is limited in its intrinsic definition, since it cannot verify the behavior of the code but only the executed statements. Mutation testing [17] represents the best

alternative to code coverage, aiming at measuring the real fault-revelation capability of a test suite. The main downside of mutation testing is that it is an extremely expensive activity. Offutt and Untch [22] grouped all the techniques developed so far to speed up mutation testing into three groups: *do fewer*, *do smarter*, and *do faster*. An application of machine learning (ML) methods to the effectiveness prediction problem was initially devised by Zhang *et al.* [41]. They proposed a classification model that predicts whether a generated mutant will be killed or not by a certain test. The work we propose to address the first research question is similar to the one of Zhang *et al.* [41]. However, their technique still requires the generation of the mutants; moreover, they rely on a series of dynamic information about code coverage and executed mutation testing, that still requires a considerable computation effort. Therefore, the work [11] we propose to answer our $\mathbf{RQ}_1$ has a different and more comprehensive goal, *i.e.*, i) predict test-case effectiveness without any dynamic information or previous mutation analysis and ii) understanding the key factors that affect effectiveness.

Test Quality. Beck [3] firstly argued how desirable is for test cases to respect good design principles such that they result easier to comprehend, maintain and successfully used to diagnose problems in the production code. One of the milestone work on the topic is the one of van Deursen *et al.* [35]. They defined a catalogue of 11 poor design solution to write tests, called *test smells*, along with refactoring operations aimed to their removal. This catalogue has been then extended by Meszaros [20], introducing, amongst the others, the *Single-Condition Tests* principle. Many work showed that test smells are effectively spread in practice [13]. Bavota *et al.* [2] studied the diffusion of test smells in 18 software projects and their effects on software maintenance. They found that more than the 80% of the investigated jUnit classes were smelly, *i.e.*, affected by at least one smell. Moreover, they showed that their presence has a negative impact on the understandability of developers. Given the relevance of the problem, different approaches have been proposed to detect test smells. Greiler *et al.* [13] presented a tool, named TestHound, able to identify smells such as *General Fixture* or *Vague Header Setup*. Van Rompaey *et al.* [36] devised code-metrics based heuristics to identify two types of test smells. Palomba *et al.* [26] more recently explored the usage of textual information from test code to detect smells, showing promising results. Palomba at Zaidman [25] investigated the extent to which test smells can be exploited to locate flaky tests. They showed that large part of the flaky tests contains smells and that their remotion both fixes the smell and the flakiness. Spadini *et al.* [34] studied the relation between test smell and change- and fault-proneness of test code. They demonstrated how smells have strong effects on test code maintainability.

Automated Test Generation. Test-data generation has been intensively investigated over the last decade [19]. Different tools have been proposed with the main goal of automatically generating tests with high code coverage [9, 19, 27, 32]. Recently, several works investigated *non-coverage* aspects in addition to reaching high coverage. Lakhotia *et al.* [18] considered dynamic memory consumption as a further objective to optimize together with branch coverage. Ferrer *et al.* [7] proposed a multi-objective approach considering at the same time code coverage and oracle cost. Afshan *et al.* [1] looked at code readability as a secondary objective to optimize and used natural language models to generate tests having readable string inputs. In all aforementioned works, coverage and non-coverage test properties were considered as equally important objectives for the evolutionary search. However, empirical studies showed the difficulty of balancing the two contrasting objectives and avoiding negative effects on the final branch coverage [7]. Furthermore, these prior studies used a classical single-target approach rather than more effective multi-target strategies. Palomba *et al.* [24] proposed an early attempt to combine coverage and non-coverage criteria within a multi-target approach. They incorporated test-cohesion and test-coupling metrics as secondary objectives within the *preference criterion* of MOSA, with the goal of producing more maintainable test cases: the result is more-cohesive and less-coupled test cases, as well as higher coverage. The work proposed in $\mathbf{RQ}_3$ builds on top of such approach by defining an adaptive strategy that enables and disables the secondary objective during the generation.

7 CONCLUSION AND FUTURE RESEARCH

In my research, I aim at making mutation testing more scalable by estimating test case effectiveness relying on source-quality-indicators. Moreover, I want to complement the usage of code coverage as a main metric to evaluate test quality by introducing a novel metric that measure test *focus* as a proxy for the effort needed by developers to understand a test case. Similarly, I plan to devise an adaptive approach that optimize any quality factor, along with code coverage, in the context of automated test case generation. The expected contributions of my research include (i) understanding of the source-code factors that impact test-case effectiveness, (ii) a ML model able to predict test effectiveness in a lightweight fashion relying on the aforementioned source-code factors, (iii) a new metric based on the concept of entropy, (iv) a proof-of-concept implementation of an adaptive test-case generation algorithm that allows to optimize any non-coverage criteria that goes in contrast with the code coverage as a main objective. While (i) and (ii) are mainly covered, my current research focuses on the definition and implementation of the latter.

I envision to extend existing tools for continuous inspection of software quality (*e.g.*, SonarQube [38]) allowing developers to diagnose the health status of test suites. From the outcome of $\mathbf{RQ}_1$, they can become aware of the poor effectiveness of some tests. Moreover, I plan to extend the effectiveness model by introducing a feature that suggests the specific characteristics that contribute for a test to be classified as non-effective. Thus, form the outcome of $\mathbf{RQ}_2$, developers can be notified of the poor *focus* of some tests, adopting preventive actions that will lead to increase their future understandability and maintainability. Finally, I envision a continuous automated test case generation mechanism that dynamically select the quality criteria to optimize as a second objective, based on the observation of the current state of the test suite.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Harald Gall for the supervising and the support. The research leading to these results receives funding from the Swiss National Science Foundation (SNF) under project SURF-MobileAppsData (no. 200021_166275). I further like to thank the Swiss Group for Software Engineering (CHOOSE) for providing financial support to attend the conference

REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 352–361, Washington, DC, USA, 2013. IEEE Computer Society.

[2] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. W. Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[3] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[4] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[6] W. Conover. *Practical nonparametric statistics*. Wiley series in probability and statistics: Applied probability and statistics. Wiley, 1999.

[7] J. Ferrer, F. Chicano, and E. Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Softw. Pract. Exper.*, 42(11):1331–1362, Nov. 2012.

[8] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[9] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, Feb. 2013.

[10] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, 66(3):854–874, 2017.

[11] G. Grano, F. Palomba, and H. C. Gall. Lightweight assessment of test-case effectiveness using source-code-quality indicators. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[12] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 348–351, New York, NY, USA, 2018. ACM.

[13] M. Greiler, A. Zaidman, A. v. Deursen, and M.-A. Storey. Strategies for avoiding text fixture smells during software evolution. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 387–396. IEEE Press, 2013.

[14] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[15] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[16] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[18] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105, New York, NY, USA, 2007. ACM.

[19] P. McMinn. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 153–163. IEEE, 2011.

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

[21] M. Mirzaaghaei, F. Pastore, and M. Pezzè. Automatic test case evolution. *Software testing, Verification and Reliability*, 24(5):386–411, 2014.

[22] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[23] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 5–14. ACM, 2016.

[24] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. Automatic test case generation: What if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 130–141. ACM, 2016.

[25] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 1–12. IEEE, 2017.

[26] F. Palomba, A. Zaidman, and A. De Lucia. Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 311–322. IEEE, 2018.

[27] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[28] A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018.

[29] A. Panichella, F. M. Kifetew, and P. Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 104:236–256, 2018.

[30] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 33. ACM, 2012.

[31] J. M. Rojas and G. Fraser. Is search-based unit test generation research stuck in a local optimum? In *Proceedings of the 10th International Workshop on Search-Based Software Testing*, pages 51–52. IEEE Press, 2017.

[32] S. Scalabrino, G. Grano, D. D. Nucci, R. Oliveto, and A. D. Lucia. Search-based testing of procedural programs: Iterative single-target or multi-target approach? In F. Sarro and K. Deb, editors, *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, volume 9962 of *Lecture Notes in Computer Science*, pages 64–79, 2016.

[33] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[34] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE, 2018.

[35] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[36] 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. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.

[37] A. Vargha and H. D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal on Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[38] C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall. Continuous code quality: Are we (really) doing that? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 790–795, New York, NY, USA, 2018. ACM.

[39] W. E. Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

[40] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai. Effective fault localization using code coverage. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 449–456. IEEE, 2007.

[41] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, 2018.