Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality

Giovanni Grano,[1] Cristian De Iaco,[1] Fabio Palomba,[2] Harald C. Gall[1]
[1]SEAL Lab - University of Zurich, Switzerland — [2]SeSa Lab - University of Salerno, Italy
grano@ifi.uzh.ch, cristian.deiaco@uzh.ch, fpalomba@unisa.it, gall@ifi.uzh.ch

***Abstract*—Test cases are an essential asset to evaluate software quality. The research community has provided various alternatives to help developers assessing the quality of tests, like code or mutation coverage. Despite the effort spent so far, however, little is known on how practitioners perceive unit test code quality and whether the existing metrics reflect their perception. This paper aims at addressing this gap of knowledge. We first conduct semi-structured interviews and surveys with practitioners to establish a taxonomy of relevant factors for unit test quality and collect a dataset of tests rated by developers based on their perceived quality. Then, we devise a statistical model to measure how the metrics available in literature reflect the perceived quality of test cases. The findings of our study show that readability and maintainability are the key aspects for developers to diagnose the outcome of test cases and drive debugging activities. On the contrary, code coverage metrics are necessary but not sufficient to evaluate the capability of tests. Finally, we discover that available metrics are effective in characterizing poor-quality tests, while limited when distinguishing high-quality ones.**

***Index Terms*—Software testing; Code Quality; Empirical SE.**

I. Introduction

During software evolution, developers perform multiple changes to the codebase to enhance existing features, implement new ones, and fix emerging defects [41]. In this context, they periodically run test cases to verify that new changes do not introduce regressions [53]. Both development and selection of tests to be run are oftentimes driven by their ability to actually catch defects in the codebase [2, 37, 64]. As it is not possible to know *a-priori* the fault detection capabilities of tests, researchers have been devising metrics to estimate the quality of tests. Among all, code coverage, *i.e.*, the amount of production code exercised by a test, is considered the main indicator for test code quality [11, 38] and, indeed, researchers and tool vendors used it to assist practitioners during all the activities connected to testing, from test selection to analysis of production code fault-proneness [2, 40, 76]. However, code coverage only indicates which part of the production code is exercised, failing to provide indications on whether its intended behavior is actually tested: as an example, Ellinms *et al.* [20] found faults in projects having a high code coverage.

The main research alternative to code coverage is represented by mutation testing: in this case, defects are artificially seeded into the production code to quantify the ability of tests to find them [49]. While this is considered as a more powerful metric for test case quality [35], its practical usage is still under debate [28, 33]. Other devised metrics, *e.g.*, test flakiness [19, 43] or test smells [46], have been also connected to software quality as well as test code effectiveness.

Recognizing the effort spent by the research community in devising test code quality metrics, we identify a common limitation with respect to the way the usefulness of these metrics has been evaluated. Most of them, indeed, have been experimented *in-vitro* with empirical studies aiming at measuring their correlation with the fault-proneness of production code (*e.g.*, [2, 28, 35]). On the contrary, there is still a lack of knowledge on (i) how practitioners define unit test code quality and (ii) whether the existing metrics match their perception, hence being considered useful in practice. An improved understanding of these aspects would be beneficial to comprehend the way the research community supports practitioners and if additional instruments would be worthwhile.

In this paper, we propose a mixed-method research approach in order to (i) elicit a taxonomy of factors deemed relevant by practitioners for unit test code quality and (ii) understand how the metrics defined in literature match the developer's perception of unit test code quality. We start our investigation by interviewing five software testing experts to let them elaborate a set of factors influencing the quality of test cases. Once established an initial taxonomy of these factors, we run a survey study that involves 70 practitioners in which we (i) evaluate the taxonomy on a larger-scale and (ii) ask them to rate 210 test cases, overall, according to their perception of quality. As a final step, we compute a number of state-of-the-art test code quality metrics on the built dataset to study the extent to which they are statistically related to what developers perceive as test code quality.

The key results of our study first report that code and mutation coverage are necessary but not sufficient indicators of the quality of tests. We discover that test code design-related attributes, like readability and maintainability, can better pinpoint test cases useful to discover defects in production code. Finally, metrics defined in literature only partially align with the developer's perception of test code quality. As such, we conclude that a novel, more comprehensive set of test code metrics should be devised to better assist practitioners when dealing with development and assessment of test cases.

To sum up, this paper provides three main contributions:

1) A novel taxonomy of factors deemed important by practitioners for assessing test code quality;
2) A statistical study of how currently existing metrics support the developer's needs in test code assessment;

3) A research roadmap that researchers should follow to better fit the practitioner's needs.

II. Background and Related Work

Test code quality represents a multi-faceted concept able to express how useful a test will be for developers during the understanding of the production code [61], the debugging activities [34, 75], and the early catching of defects [26]. Over the last decade, a number of researchers have been studying test code quality with the aim of defining metrics able to characterize it under different perspectives.

In the first place, code coverage has been widely used in practice to assess the quality of test suites, as it is easy to compute (*e.g.*, by continuous integration tools) and easy to interpret [69]. Similarly, a large body of research focused on the relationship between code coverage and test quality and, in particular, previous work investigated the role of code coverage in fault localization [73, 74] and detection [11]. However, code coverage has been shown to be an insufficient indicator when it comes to assess test quality [32]. Rojas and Fraser [56] stated that its main limitation is the inability to verify the intended behavior while merely looking at the execute code. Mutation testing, based on the idea of *mutants*, represents the best alternative to code coverage [3, 33, 35]. However, it still suffers from scalability issues—despite the attempts done by researchers to alleviate this problem [28, 49, 77].

On another note, Beck [7] suggested that good design principles, understandability, and maintainability are desirable properties of test cases. For this reason, researchers [46, 66] devised catalogs of poor design solutions named *test smells*. Bavota *et al.* [5] investigated the diffusion of test smells in large open source projects showing how their presence has a negative impact on program comprehensibility and maintainability. Along this line, Spadini *et al.* [62] studied the relationship between smells and change- and fault-proneness of both test and production code, reporting similar findings.

Another perspective of test code quality relates to the role of assertions. Kudrjavets *et al.* [39] showed a relationship between assertion density, defined as the number of assertions over the KLOC of a class, and the decrease of faults in production code. Their main finding is the inverse relationship between bugs and assertion density, *i.e.*, the higher the number of assertions, the lower the fault-proneness of production code. Similarly, Hoare [31] focused on the number of pre- and post-conditions in the context of 21 different software projects, showing a positive correlation between their number and the stability of those projects. Moreover, previous work studied the relationship between the usage of assertions and experience of software programmers [12] reporting that experienced developers tends to rely more on assertions.

Source code readability is an important property when it comes to perform maintenance and evolution tasks [42]. Previous research showed that readability metrics correlate to the fault-proneness of source score [45, 58]. Grano *et al.* [28] investigated this aspect in the context of test code, showing that developers tend to neglect the readability of test cases. In so doing, they relied on the state-of-art model for readability [59]: being trained on both production and test code snippets, it is suitable to analyze test readability as well [29].

From an empirical viewpoint, Nagappan *et al.* [47] were the first trying to capture the quality of test suites using a variety of product and assertion-related metrics. While sharing a similar long-term objective, our goal is to better analyze the practitioner's perspective with respect to unit code quality in order to understand how they assess tests in practice. Finally, the closest related work is the one by Bowes *et al.* [8]. The authors reported the results of a two-day workshop with practitioners in which they elicited a set of testing principles that not only address code coverage-related metrics, but also other quality facets of testing. As a result, they identified 15 principles that range from keeping maintainability into account to the need for considering happy and sad paths when testing production code. This work can be considered as complementary since we aim at gaining a broader understanding of how developers perceive unit test code quality and how the metrics defined in literature match this perception.

III. Research Goals and Questions

The *goal* of the empirical study is to (i) elicit the factors deemed important by practitioners when assessing the quality of unit tests and (ii) understand how the test code quality metrics defined in literature align with those considered relevant in practice. The *purpose* is to study if additional, complementary test code metrics should be devised or whether the assistance currently provided is sufficient. The *perspective* is mainly that of researchers interested in analyzing the support they currently provide to developers with respect to test code metrics. Our study is structured around two main research questions. We start by focusing on the *practitioner's perspective* of test code quality, trying to investigate and extract a set of features that developers consider relevant when assessing the goodness of unit tests. Hence, we ask:

> **RQ_1:** *What are the features of unit tests that, according to developers, have an influence on unit test quality?*

Afterwards, we turn our attention on the *research perspective* of test code quality, namely we analyze the support that is currently provided by the research community with respect to the assessment of test code quality as well as the alignment between the metrics proposed in literature and the features deemed important by developers in practice:

> **RQ_2:** *Do existing test code quality metrics match the developer's perception of test code quality?*

To address the two research questions, we feature a mixed-method research approach [17] that combines insights from semi-structured interviews and surveys with statistical results investigating how existing test code quality metrics match the developer's perception of test code quality.

IV. RQ_1. The Practitioner's Perspective

As a first part of our investigation, we study what developers perceive as important when it turns to test code quality.

A. Research Methodology

To address our first research question, we need to capture a broad variety of practitioner's opinions on test code quality.

Semi-structured Interviews. First, we conduct semi-structured interviews with software testing experts in order to start elaborating an initial taxonomy reporting the factors that should matter when assessing the quality of unit tests. This research approach is often used in exploratory investigations to understand phenomena and seek new insights [70]. In our case, we decide to start with semi-structured interviews as we prefer letting emerge possible factors influencing test code quality directly from the opinions of experts rather than from our own view of the phenomenon: this reduces the introduction of possible sampling and inclusion biases [60], other than favoring the emergence of factors actually used by practitioners in their daily development activities.

The general structure of the interviews is composed of three parts. After some background questions aimed at characterizing the sample of the involved practitioners, the first part consists of a general discussion on the practices applied when developing unit tests, with a particular focus on (i) the granularity adopted to create unit tests (*e.g.*, if they develop test cases targeting specific production methods or follow a different approach) and (ii) the tools used to assess state and quality of test cases. In the second part, we discuss about the developer's definition of a high quality unit test. Once provided a high-level interpretation, we ask the interviewee to show and describe us one of her/his unit tests which s/he deems to be of a high quality: in so doing, we expect the interviewee to provide further and finer-grained insights into the aspects making unit tests good. Finally, in the last part of the interview we ask the participants to summarize their thoughts on high quality unit tests into measurable factors or software engineering methodologies that may possibly assess or foster test case quality. The three parts, altogether, aim at contributing to the construction of an initial taxonomy of factors influencing test code quality.

After designing the structure of the interviews, we define the recruitment strategy. Ours can be considered as a *convenience* sample [36], in which we invite five software testing experts from our personal industrial contacts. One of them hold a Bachelor degree in Computer Science, two a Master degree, and the remaining two a Ph.D. degree in Software Engineering. Overall, they have between 3 and 10 years of experience in testing and typically develop multiple unit tests per day.

The semi-structured interviews have a duration from 40 to 60 minutes and are conducted between December 2019 and January 2020 through an either face-to-face meeting or remote Skype call in which at least two of the authors of this paper participate. All interviews are recorded and then transcribed for analysis, preserving the anonymity of the interviewees. We share these transcripts in our appendix [1].

The collected data are then analyzed by the first two authors of this paper adopting the following methodology:

Step 1 - Summarization: Initially, one inspector summarizes the semi-structured interviews and groups the available pieces of information into three categories: (1) *'Applied practices'*, (2) *'Definition of unit test code quality'*, and (3) *'Possible features to compute it'*. These correspond to the three main parts of the interviews.

Step 2 - Microanalysis: The same inspector starts extracting relevant pieces of information and assigns temporary labels that represent concepts emerging from the interviews that may be relevant for the assessment of test code quality.

Step 3 - Categorization: The two inspectors jointly analyze the labels assigned in the previous step in order to cluster those that are semantically similar or even identical [30]. This step also allows the second inspector to double-check the operations done in the previous steps.

Step 4 - Saturation: The two inspectors iterate over the labels assigned so far until they can reach a full agreement with respect to names and meanings of all of them. This step leads to a theoretical saturation [68], namely the phase where the analysis does not propose newer insights and all concepts expressed by the interviewees are well-developed.

Step 5 - Taxonomy Building: Based on the labels assigned, the two inspectors proceed with the construction of the initial taxonomy, *i.e.*, they specify the factors deemed important for test code quality by the interviewees.

Confirmatory Survey Study and Dataset Collection. While the semi-structured interviews lead to an initial taxonomy of factors contributing to test code quality, we conduct a larger-scale survey study aimed at (i) confirming the validity of the initial findings, (ii) suggesting additional factors not covered by the interviews, and (iii) building a dataset of unit test cases rated by developers according to their perceived quality.

The survey is composed of three main sections—for the sake of space limitations, we report the full list of questions included in the survey in our appendix [1]. In the first one, we ask the participant's opinion on the features that most influence test code quality: particularly, we not only seek opinions on the features considered relevant, but also on whether they are effectively measured in the working environment of the participants and, if so, how. In this stage, we allow participants to report a maximum of six factors each. In the second section, instead, we propose the source code pertaining to three unit test cases, along with information about their code and mutation coverage, and ask participants to rate them based on (i) their overall quality and (ii) the features mentioned in the first section of the survey. The test cases proposed to each participant are randomly selected from a pool of ten open-source tests that we mine from systems of the APACHE ecosystem. We only proposed three tests to avoid having an excessively long survey which may have resulted in an increase of the abandonment rate [21]. In particular, before running the survey we first extract all unit test cases (along with the corresponded exercised classes [28]) belonging to APACHE COMMONS. Then, we randomly pick ten distinct tests coming from different suites: these tests form the pool used

in the survey [1]. The selection of APACHE COMMONS is based on two main reasons: first, it contains a set of libraries that are widely used by practitioners worldwide [54], possibly letting survey participants to know (or to be able to acquire knowledge on) the considered systems; second, it contains a large amount of test cases written by hundreds of different contributors [25], hence increasing the diversity of the unit tests analyzed. It is important to note that we cannot provide developers with tests developed by their own as the survey is intended to be disseminated at large scale. Overall, we collect 210 evaluations that are used later in the context of RQ_2. We prefer collecting multiple ratings for a pool of unit tests rather than individual scores on a larger amount of tests because in this setting we can also analyze the variance of the ratings and provide insights into the agreement reached on the sample tests. Finally, the last section of the survey aims at collecting background information on the participants.

The survey is implemented using LIMESURVEY.[1] It is made available from March 1st to 31st, 2020 and advertised through personal contacts and social network accounts of the authors, *i.e.*, FACEBOOK, TWITTER, REDDIT, and LINKEDIN.

All in all, we receive 70 fully compiled questionnaires. The same authors who were involved in the analysis of the semi-structured interviews analyzed the survey responses. In so doing, they apply exactly the same methodology described for the analysis of the semi-structured interviews when considering the factors emerged from the surveys. The only additional step performed in this case is the activity of merging the factors emerged from the survey with those highlighted by the interviews in case the same concepts were expressed. The final outcome consists of a validated taxonomy of factors influencing test code quality, which we describe in the following section. As for the collected dataset, this is described and analyzed when addressing RQ_2 (see Section V).

B. Analysis of the Results

This section discusses the main findings for RQ_1.

A taxonomy of unit test quality features. Five testing experts were initially interviewed with the final aim of deriving a set of features that impact on the quality of unit tests. In the first place, however, it is worth to briefly discuss the main outcomes coming from the analysis of the practices they typically use to develop test cases as well as the tools employed to assess state and quality of the test suites. As a matter of fact, all our interviewees declared that they prefer adopting an approach which is mostly inspired to test-driven development [7], meaning that they like to start writing test cases before production code or, at very least, proceeding with a *test-as-you-write* strategy, *i.e.*, they develop production and test code in parallel. The testing experts revealed that such a strategy typically allow them to spot edge cases first, being therefore able to produce higher-quality production code. Hence, our findings seem to confirm previous quantitative results showing the positive effects of test-driven development on source code quality [44, 55]. Furthermore, our interviewees reported that they typically start creating test cases by focusing on individual use case scenarios of the production code, only later extending the test suites to incorporate additional scenarios: for example, interviewee #5 explained that s/he starts testing by developing one single scenario, with a single assertion, per test.

Secondly, our interviewees reported the use of a wide range of testing frameworks, from test automation (like testing in continuous integration pipelines) to mocking frameworks usable to effectively isolate the scope of unit tests. They pointed out that these tools are necessary to enable the creation of effective tests having a high fault detection capability.

Finally, Table I overviews the features characterizing unit test code quality as well as how to measure/deal with them according to the opinions of our interviewees. As shown in the table, the extracted features could be classified into three main categories such as *'Behavioral'*, *'Structural'*, and *'Executional'*, which we further analyze in the following.

Behavioral features. The first category is composed of four macro-factors that relate to the nature and behavior of unit tests. According to the involved testing experts, one of the key characteristics making a test of high-quality is its ability of being (self-)validated: this implies that the test should neither be defective nor require additional checks (*e.g.*, other pieces of code) to be verified. This aspect, besides being somehow expected to be critical for the development of good tests, is actually one of the F.I.R.S.T. (Fast, Independent, Repeatable, Self-validating, Thorough) principles which originally inspired born and rise of test-driven development [7]. The (self-)validation of test cases was mentioned by three of the experts, who all reported code review as the software engineering practice that could assist the assessment of test cases—thus, suggesting that tests should also be part of the code review process [61].
The scope of a test was the factor mentioned by all interviewees. This refers to the extent to which the behavior of the unit under test is actually exercised by a test. In other words, a critical factor for developers concerns with the ability of assessing whether a unit test actually exercises the corresponding unit and, if so, how many use case scenarios it covers. Accordingly, the first metric mentioned was code coverage, *i.e.*, how many lines of production code are touched by a unit test. Nevertheless, the experts also pointed out that the complexity of a test plays a role in this case: indeed, the higher the complexity of the test, the lower the developer's ability to understand its scope, and therefore its target. In a complementary manner, the experts revealed the estimation of the effectiveness of a unit test as a critical factor to consider. In so doing, all experts mentioned code coverage as a metric to use for this purpose. However, all of them agreed on the fact that this is just to consider a proxy measure that is *necessary* to use, but definitively *not sufficient*. To make her/his reasoning more practical, interviewee #3 reported the case of a critical system:

"So, if we are building a critical system, then you

[1] Link: https://www.limesurvey.org

TABLE I
TAXONOMY OF UNIT TEST QUALITY FEATURES AND CORRESPONDING MEASURABLE FACTORS/PRACTICES.

Category	Sub-Level	Description	Measurable factors/practices
Behavioral	(Self-)validation	A test should behave as expected, *i.e.*, it must not be defective	Test code review
	Scope	Extent of the code under test exercised by a single unit test	Code coverage; Test case complexity
	Effectiveness	Ability of revealing fault in the exercised production code	Code coverage; Mutation coverage; Purpose
	Diagnosability	Features that facilitate fault detection and solving	Comments to assertions; Failure reproduction; Test code review
Structural	Size	Size of the unit test case	Lines of Code
	Test Design	Features about the general structure of a test	Assertion density; Comments to assertions; Test code complexity; Lines of Code
	Reusability	Reusability of unit tests in other suites	
	Readability	Readability of the test code	Readability
	Maintainability	Maintainability and evolvability of the test code.	Test smells; Time required to fix an assertion
	Independence	Degree of isolation of a unit test with respect to the other tests of the suite	Coupling metrics, like CBO [14]
Executional	Execution Time	Time taken by a unit test to be executed	Execution time
	Reliability	Unit tests should always produce the same results	Test code flakiness
	Execution Infrastructure	Availability of information about the execution environment of a test	Statistics of Continuous Integration servers

should have a decent branch coverage, because I mean it is a critical system. So, you really need to test it very well. And in that case, branch coverage is not really a good metric of how good your test suite is, because I can easily have 100% test coverage."

The experts reported the lack of alternative metrics able to provide more insightful indications of how effective a test actually is. Only one of the experts reported to use mutation coverage, *i.e.*, the number of artificially created defects that a test can find [50], in specific cases to better understand the effectiveness of tests—somehow confirming the limitations of mutation analysis in practice [3, 18, 28]. Finally, the *purpose* of the test, *i.e.*, the requirement that a unit test is exercising, was suggested by two experts as an ideal metric to verify that a unit suite can actually exercise the corresponding code in a thoroughly manner.

Last but not least, all experts agreed that the diagnosability of the test outcome is key to enable the detection and fixing of faults. In this case, they suggest that test code documentation and, in particular, the addition of comments to assertions can substantially help understanding why a test fails. The reproducibility of a test is also a factor deemed to be relevant: two of the experts referred to test flakiness [43] as a metric to use to estimate how reproducible unit tests are. Interviewee #1 recommended test code review as a practice to spot possible threats to test reproducibility.

Structural features. This category collects the factors that concern with the internal structure of a unit test. According to our experts, structural aspects can influence not only the understandability of a unit test, but also the overall resulting effectiveness in both fault detection and diagnosability. For example, they suggest that keeping the size low is a good way to create concise tests that ease the comprehensibility of the target behavior of the production code. Similarly, the experts expressed the need for readable test code which can be quickly interpreted in case of failure as well as maintainability properties that enable tests to be evolved in an easier manner: in these cases, they also named specific metrics such as readability [10] and test smells [46], respectively. Interestingly, we noticed that all experts were aware of the concept of test smells and their potential negative effects. While this seems to be in contrast with previous work reporting that developers cannot often recognize smells in test code [65], our findings suggest that experience matters and that developers used to develop test cases are more sensible to design issues and can recognize them as a threat. Besides the factors discussed above, all interviewees reported that test design is crucial for the development of high quality tests. According to them, producing tests having a good assertion density, *i.e.*, the number of assertions per test case size [39], is important but, at the same time, these assertions should always be accompanied by some form of documentation that can clearly point out which of them fails as well as the reason behind the failure. This aspect is strictly connected to the *Assertion Roulette* test smell, which appears when a test includes a number of assertions without comments [46]. Finally, avoiding the creation of complex tests reduces the cognitive load needed to understand them. The independence of unit tests was also named by three experts. This basically refers to having poorly coupled tests that do not interact between them. In this respect, interviewee #4 reported that this reduces the risk of interference of some unit tests toward others, which can normally cause forms of test flakiness [43]. Finally, interviewee #2 mentioned reusability: in her/his opinion, having the possibility to reuse tests in other suites leads to two benefits: (i) it reduces the risk due to a new implementation and (ii) increases the chance of relying and evolving effective tests

TABLE II
DEMOGRAPHIC INFORMATION OF SURVEY PARTICIPANTS.

Education			Experience (years)		
High School or eq.	12	17.14%	1-5	24	34.29%
Bachelor	28	40.00%	6-10	15	21.43%
Master	17	24.29%	11-20	10	14.29%
Ph.D	8	11.43%	20+	5	7.14%
Other	4	5.71%			
Current Employment			**Team Size**		
Student	15	21.43%	1-4	12	17.14%
Researcher	6	8.57%	5-10	22	31.43%
Prof. (paid) Dev.	55	78.57%	11-20	10	14.29%
Prof. (unpaid) Dev.	1	1.43%	20+	10	14.29%

that can be used to catch defects in various parts of the code. Nonetheless, the expert could not report a possible metric or practice that could help measuring reusability.

Executional features. The last category refers to the execution of the tests. While all experts agreed that the reliability of a unit test is a relevant factor to avoid flaky tests, *i.e.*, intermittent tests that pass and fail with the same code [43], only interviewee #5 suggested two additional aspects. First, s/he reported that the execution time of a unit test should be kept low to have a quick feedback on the production code quality. Second, the availability of information about the infrastructure environment could provide additional knowledge of whether a test risks to be flaky.

Insights from the survey. Once created an initial taxonomy of factors contributing to unit test code quality, we proceeded with its larger-scale validation through a survey study which involved 70 developers worldwide. Table II summarizes descriptive statistics of the survey participants: most of them are currently professional developers working in teams composed of 5-10 members and with an experience of up to 5 years. It is worth noting that we did not make background questions mandatory in the survey since some participants might not feel comfortable with providing information about their status [16]; hence, the table includes data concerning the participants willing to fill the background part out.

Figure 1 shows the results achieved when inquiring the participants on the features they consider important for unit test code quality. We could immediately see an almost perfect match between their opinions and the initial taxonomy extracted through the semi-structured interviews. Indeed, after merging the factors named in the survey with those mentioned by the testing experts, we discovered that the survey participants had comparable thoughts when discussing test code quality. For instance, the need for understanding the scope of a unit test was mentioned 76 times by our survey participants, *i.e.*, some of them named multiple times characteristics falling under the *"Scope"* sub-level. Survey participants named much more frequently metrics related to structural properties of unit tests rather than those belonging to other categories.

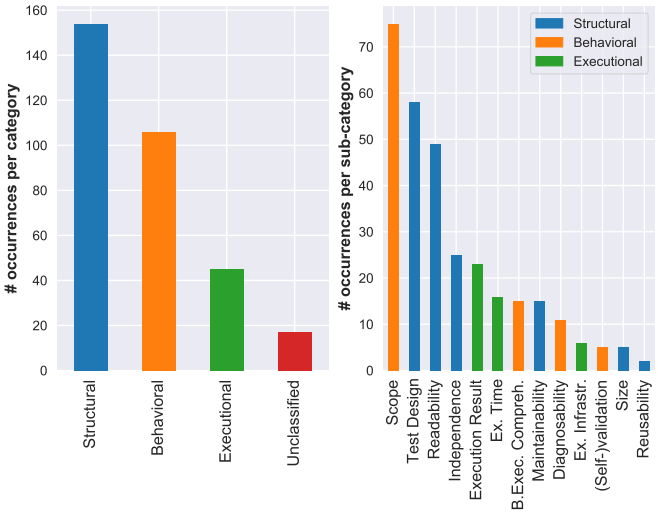

Fig. 1. Number of occurrences of the factors deemed important in the survey for each category and subcategory of the taxonomy.

This result suggests that test code design is among the most pressing contributors to unit test quality, as confirmed by the amount of characteristics falling under the *"Test design"* and *"Independence"* sub-levels named by the survey participants. At the same time, maintainability and readability aspects are considered important as well: as a matter of fact, the *"Readability"* sub-level represents the third most frequent aspect mentioned in the survey.

A few exceptions to this general discussion were also registered. In 18 cases, survey participants defined characteristics that could not be assigned to any of the categories of the initial taxonomy—column "UNCLASSIFIED" in Figure 1. These answers, however, did not provide any additional aspect to consider for unit test code quality but rather reported either too generic considerations (*e.g.*, one participant explained that tests *"should be treated as a first-class citizen just as production code"*) or the description of philosophies to use when developing source code (*e.g.*, one of them suggested test-driven development when inquired about the characteristics of high quality unit tests). As such, those answers could not be considered for extending the taxonomy. To conclude this first analysis, we can therefore say that the initial taxonomy was extensive enough to be considered complete by the larger crowd of developers involved in the survey study.

> **RQ_1 Summary.** In the first place, RQ_1 confirms the idea that test code quality represents a multi-faceted concept which is composed of a number of different aspects and characteristics that can, to some extent, be measured. Not only tests should effectively exercise the production code and be able to detect faults, but developers tend to be much more focused on non-functional aspects of test code. Indeed, it is possible to delineate a trend in the answers of testing experts and survey participants: *readability and understandability of tests are key factors, perhaps even*

more important than their ability to cover specific paths of the production code. These characteristics enable a finer-grained verification of the executed paths and would allow to have a clearer idea of what tests should be added to properly exercise the source code; this is not always the case of quantitative code coverage metrics, which cannot "explain" and make immediately "comprehensible" the production code that is exercised. In the second place, our findings confirm that test code design and smells play a role toward the making of meaningful test cases.

V. $\mathbf{RQ}_2$. The Research Perspective

In $\mathbf{RQ}_2$ we exploit a statistical model relating existing metrics to the developer's perception of unit test code quality.

A. Research Methodology

Response Variable Definition. It is represented by 210 scores about perceived code quality given to 10 test cases (fully detailed in [1]) by the participants of the study. We presented to each participant three of them, along with the correspondent production classes to give them the necessary context for their understanding. The participants rated them by selecting a value on a Likert scale. They also had the possibility to leave the question blank or to answer with *"I do not know"*. After cleaning the results of the survey by discarding such cases, we ended up with 199 different evaluations lying on a Likert scale from 1 to 5, representative of the following values: *very poor*, *poor*, *fair*, *good*, and *very good*. We report in [1] the descriptive statistics for the collected evaluations, describing for each rated test the mean of the attributed scores along with the median, the standard deviations and the total number of evaluations. From their analysis, we could observe that the scores given to the tests have a standard deviation of $\approx$1, meaning that the developers tend to assess the quality of the unit tests in a similar manner—hence, we have a homogeneous dataset that does not present outliers and that can be actually useful to understand which are the features related to the general perception that developers have of unit test quality.

Independent Variables Definition. We use the outcome of RQ_1 to establish a set of computable metrics that may relate to the dependent variable. This leads to the definition of 11 metrics, described as follow.

Mutation score. This is the ratio between the mutants, *i.e.*, artificially created defects seeded in production code, effectively detected by the test over the total number of generated mutants [50]. This is considered the high-end criteria when it comes to measure test code quality [33] and, according to the results of RQ_1, testing experts suggested that this metric could be actually used to assess test effectiveness. We compute the mutation score relying on PIT [15]. This choice is due to the fact that PIT has been employed in previous research about mutation testing [28, 32, 78]. Moreover, it represents the most reliable and mature mutation testing engine freely available [18].

Code coverage. This metric is largely used in practice to assess test code quality and report immediate feedback to developers [69]. The testing experts interviewed in RQ_1 reported that it may be useful to measure scope and effectiveness of unit tests. In this work, we use line coverage, *i.e.*, the percentage of lines of code covered on the production code by the execution of a test, computing it using PIT.

Code metrics. We select a set of 5 code metrics related to code complexity (RFC, WCM, NOSI), coupling (CBO) and size (LOC). We include these metrics because (i) testing experts reported them to be potentially useful for test code quality and (ii) they capture various structural aspects of source code that may contribute to the response variable. To compute those metrics, we rely on the `ck` tool developed by Aniche [4]. It is worth to remark we calculate those metrics at method level, *i.e.*, uniquely on the test methods included in the study and rated by the participants.

Test smells. Test smells are sub-optimal implementation choices related to tests [46, 65, 66]. Previous research showed that they affect maintainability and effectiveness of test code [6, 62]. Based on the results from RQ_1, we first consider the *Assertion Roulette* smell [66]: this smell is detected when a test has a number of assertions *without* explanation. As such, we can capture the impact of (lack of) assertion documentation, which has been mentioned by testing experts as a relevant factor for unit test quality. Secondly, we consider *Eager Test* [66], which affects unit tests exercising more than one production method: this may affect the maintainability of tests but also intuitively hinder the identification of the scope of a test. To compute test smells, we rely on the detection tool proposed by Bavota *et al.* [5], which has been largely exploited and validated by previous studies [28, 65].

Assertion density. For a test class T_i, assertion density is defined as the number of assertions in T_i over the thousands lines of code of T_i [39]. Previous research has shown that high value of assertion density relates to fewer faults in production code [13, 39]. Furthermore, it was mentioned in RQ_1 as a potential factor contributing to test code quality.

Readability. This has been not only highlighted by testing experts in RQ_1, but also considered as one of the most important factors by the survey participants as it eases maintenance and evolution tasks [42]. To compute the readability we rely on the original implementation of the state-of-the-art model proposed by Scalabrino *et al.* [59]. This model outputs a readability score in the $[0, 1]$ range and can be used on test code, since it has been trained on both production and test code snippets [29].

From the model we exclude some of the factors mentioned in the context of RQ_1 such the executional factors and reusability. As for the former, the main reason for this choice lies in our willingness to avoid the presentation of information that developers could not directly access and assess by looking at the evaluated code, *e.g.*, they could only analyze the unit test and its corresponding production class but could not assess

its execution time or whether it had an intermittent behavior. If we would have included those pieces of information, we would have potentially risked the introduction of the so-called *extreme* bias [52], which is a type of cognitive bias where participants decide based on information they could not have access to. To further examine the role of behavioral metrics on unit test quality, our future research agenda includes the execution of a controlled study where developers actively perform tasks on unit tests before assessing their quality. As for reusability, none of the involved developers was able to provide us with a concrete metric to compute it.

Control Variable Definition. While the taxonomy proposed in this paper reflects the developer's opinions on unit test code quality, it is worth remarking that their perception and, thus, the scores they assigned to the evaluated snippets may be a reflection of their *experience* with testing software systems. In other words, it could be possible that their expertise has influenced the way they evaluated unit test code quality. For this reason, we decide to account for this aspect and consider the experience developers declare while compiling the survey as a control factor of the model: in this way, we verify the effect of developer's experience on the response variable.

Statistical Modeling. After selecting and computing the model variables, we devise a proportional odds model [72] to determine the relation between the perceived test quality and the test code metrics. Proportional odds model is a class of generalized linear model that is used to predict an ordinal variable, *i.e.*, a variable that assumes values on an ordinary scale where the ordering of the values is relevant, on a set of discrete or continuous independent values [72]. This kind of regression fits our problem, being our response variable a value on a Likert scale. More formally, let indicate with Y a possible outcome with J different categories, where $|J| \geq 2$. Let define $\gamma_j = P(Y \leq j)$ as the cumulative probability of an outcome Y less than or equal to a specific category $j \in 1, ..., J-1$. Note that $P(Y \leq J) = 1$. The general form a linear logistic model for the *j*th cumulative response can be formalized as:

$$logit(\gamma_j) = \alpha_j - \beta_j^T x$$

in which both the intercept α and the covariate coefficients β depend on the category j. Since in a proportional odds model the intercepts depend on j but the slopes are equals, the odds ratio of an outcome $Y \leq j$ can be simplified as $\alpha_j - \beta^T x$.

To implement the model, we use the `polr` function from the `MASS R` package. We check the assumption of absence of multicollinearity, occurring when two or more covariates are highly correlated to each other [48]. This might cause problems in understanding the contribution of each variable in explaining the dependent one as well as issues in estimating the coefficients of the regression [48]. To this aim, we first apply hierarchical clustering, based on the Spearman's rank correlation coefficient ρ [63], on the independent variables by using the `varclus` function from the `Hmisc R` package. Thus, we inspect pairs of variables with $\rho > 0.6$ and we exclude one of them from the model, keeping the simplest and easier to interpret for the model results.

TABLE III
RESULTS OF THE REGRESSION MODEL.

	Coefficient:		
	Estimate	Std. Error	Sig.
Assertion Roulette	−2.002	0.792	*
Assertion Density	4.165	2.649	
Readability	−2.349	1.614	
CBO	−0.944	0.462	*
WMC	−0.095	0.143	
RFC	−0.247	0.151	
NOSI	−0.162	0.071	*
LOC	0.051	0.043	
Mutation Score	2.886	1.256	*
Experience	−0.010	0.020	
	Intercepts:		
very poor | poor	−8.887	2.589	***
poor | fair	−7.380	2.562	**
fair | good	−5.926	2.524	*
good | very good	−4.418	2.253	.

Signif. codes: .p<0.1; *p<0.05; **p<0.01; ***p<0.001

B. Analysis of the Results

Table III shows the results of the proportional odds model. The table lists the independent factors, the control variable and the intercepts. For each of them, we report the estimate in the model, the standard error and the statistical significance. The statistical significance is given by the number of stars as reported in Table III: '***' indicates $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, and '.' indicates $p < 0.1$. From the hierarchical cluster analysis we discovered high correlation between line coverage and mutation score, in line with what suggested by previous research [28]. Thus, we decided to keep only the latter factor in the model. Also, we excluded *Eager Test* because all the tests were affected by this smell, hence not allowing the model to use it as an independent variable.

Proportional odds models report the covariate coefficients scaled in term of logs, making their interpretation harder. For this reason, to ease the discussion of the results we converted the coefficients into the odds ratio by exponentiating the estimates—using the `exp R` command. The resulting proportional odds ratios (ORs) have the same interpretation as the odds ratios in a binary logistics regression [72].

Looking at the results, we could find that mutation score has the highest proportional odds ratio ($OR = 17.92$) amongst the significant covariants, with a $p < 0.05$: this indicates that a higher mutation score increases the probability of having a high quality unit test, as perceived by developers. This is in line with previous research showing that mutants can be a valid mechanism to assess unit test quality [35]. As explained above, we excluded line coverage from the model because of its correlation with the mutation score. To verify the impact of

this choice on the statistical results, we experimented with an alternative model which includes line coverage as a feature and discards the mutation score instead. We observed that both this model and the original one, *i.e.*, the one including mutation score, have the same Akaike Information Criterion (AIC) estimate of 482.17 and similar odds ratios for the two metrics, meaning that line coverage has a similar correlation with the scores given by developers.

Three independent variables show an inverse effect of the perceived test code quality (all with $p < 0.05$). Specifically, higher values of CBO (Coupling Between Objects) decrease the probability of observing high quality scores ($OR = 0.39$). This somehow confirms what developers reported in the context of RQ_1. Indeed, tests with high CBO might either interact with other tests or with multiple production classes. Therefore, they might have an excessively broad scope or exercise multiple functionalities.

Similarly, the presence of the *Assertion Roulette* smell, *i.e.*, tests where assertions are not documented, negatively impacts test quality perception ($OR = 0.13$). This is again in line with the results of RQ_1, where developers reported that lack of assert documentation represents a key problem for understanding what the test is supposed to do.

Finally, a similar relationship was observed when considering NOSI (Number Of Static Invocations) ($OR = 0.85$) — this metric turned out to be highly correlated with the absolute number of assertions in a test. Interestingly, however, the assertion density variable, despite an insignificant p-value and a high standard error, had a strong positive estimate. At first glance, these two results seem to contrast each other. On the one hand, the lower the NOSI, the higher the quality. On the other hand, the higher the assertion density, the higher the quality. However, we can intuitively say that our findings suggest that the number of assertions to put in a unit test should be proportioned to its purpose. Indeed, according to our findings, too many assertions are correlated with a decrease of perceived quality but, if the assertion density is proportioned, then the resulting quality is perceived differently.

On the other side, we discovered that all other metrics (size, complexity, and readability) are not statistically related to the developer's perception of test code quality. Particularly interesting is the case of readability: despite it was mentioned multiple times as a relevant feature by developers in RQ_1, the statistical results are not aligned. This finding is likely due to the poor ability of current readability metrics, as well as proxy indicators of this aspect like complexity metrics, to capture the actual understandability of source code [51, 57]. In other words, our findings support the claim for which novel metrics should be devised to better capture both structural and conceptual aspects of test code. As a final note, the control variable selected, *i.e.*, the developer's experience, did not appear as significant, meaning that this factor did not act as confound for the response variable.

To broaden the scope of the discussion, let consider the value of the intercepts for the categories. Recall that each intercept corresponds to $P(Y \leq j)$, *i.e.*, the cumulative probability of an outcome being a category lower or equals than Y against being in categories above. As an example, the $\langle$*very poor* | *poor*$\rangle$ boundary corresponds to the probability of an outcome Y to be not better than *very poor* against being in a category from *poor* above. Looking at Table III, it is interesting to note that the estimate of each level becomes less and less statistically significant as the Likert scale increases. Indeed, we observe a $p < 0.001$ for the first level, *i.e.*, the model is able to predict a *very poor* outcome by exploiting the variables we presented. At the level $\langle$*poor* | *fair*$\rangle$, the model starts losing statistical significance ($p < 0.01$), meaning that it becomes less capable of discriminating tests whose quality was categorized between *poor* and *fair*. The statistical significance lowers even more when considering the boundaries $\langle$*fair* | *good*$\rangle$ ($p < 0.05$) and $\langle$*good* | *very good*$\rangle$ ($p < 0.1$).

> **RQ_2 Summary.** The results of the intercepts suggest an interesting finding: the main factors defined by researchers as a proxy for unit test quality succeed to discern low-quality tests from fair ones, while they are less effective in doing the same for tests of higher quality. This finding aligns with what observed in RQ_1: widely used metrics, *e.g.*, code coverage ones, are often necessary but not sufficient to guarantee high test code quality. Mutation score is the factor with the highest predicting power for high perceived quality, while high test coupling and undocumented assertions show an opposite impact. Our analysis partially confirms the usefulness of existing metrics but, at the same time, reveals a limitation: these metrics fail at providing a comprehensive and complete model of perceived test quality.

VI. Implications of the Study

The results of our study provided a number of implications for the software engineering research community.

The existing metrics are not enough. From the results coming from both RQ_1 and RQ_2 we could derive important insights into the practical relevance of existing test code quality metrics. The practitioners involved in our study first highlighted how code coverage metrics are necessary since they provide information on the amount of exercised production code. However, they are not enough to guarantee neither the diagnosability of the faults discovered nor the understandability of the code under test. Furthermore, mutation coverage is rarely applied in practice and, indeed, developers tend to assess unit test quality by using different metrics. For example, the testing experts involved in our study suggested that important facets of test code quality are currently either under-investigated, *e.g.*, readability, or not considered yet, *e.g.*, reusability. Moreover, these alternative aspects seem to be among the most important for practitioners: as a matter of fact, when inquired about the factors influencing unit code quality, the survey participants often mentioned readability and maintainability as top factors influencing unit code quality. Perhaps more importantly, from the statistical exercise of RQ_2 we discovered that code and mutation coverage, as well

as other metrics defined by the research community, *e.g.*, assertion density, effectively support developers in detecting unit tests having a low code quality. We argue it would be equally important to provide them with metrics able to better characterize high quality tests: First, such metrics could support selecting and/or prioritizing testing activities. Second, features characterizing high quality tests could provide a guideline for developer in writing more effective tests or improving existing ones. To sum up, our findings *represent a call for new metrics that can enact a more comprehensive view of the unit test code quality phenomenon, but also better characterize high quality unit tests*.

Toward explainable testing metrics. According to our findings, a key factor influencing unit test quality is represented by the explanatory power of a test, *i.e.*, by its ability to describe which use case scenario or part of the production code is exercised. Both testing experts and participants involved in RQ_1 reported the need for mechanisms fostering the explainability of a unit test. This aspect also emerged in RQ_2, where we observed that the absence of assertion documentation (*i.e.*, the presence of the *Assertion Roulette* smell) was one of the few significant factors of our statistical model. These findings directly impact the research community and, more particularly, the way testing metrics should be presented to developers. In the first place, quantitative metrics should be combined with summarization mechanisms able to describe them as well as their effects on production code. As such, our study *promotes and further stimulates the research done on source code summarization* [24]. At the same time, our findings also *stimulate the research around the under-investigated field of test code refactoring*: in particular, automated solutions able to recommend appropriate assertion descriptions could be worthwhile to improve the overall test code quality.

Design for test code quality. The ability of designing high quality test cases is considered relevant by practitioners. According to the findings of RQ_1 and RQ_2, this is especially true when considering specific aspects like test coupling and complexity. Besides the definition of novel metric able to better model these aspects, our study leans toward the definition of more structured methodologies to develop and maintain test cases. This represents a crucial challenge for the research community. While our findings *motivate the growing research area around test code design* (*e.g.*, how to best generate tests automatically [22, 23, 27]), we believe that further efforts should put in place for the *definition of best and bad practices that can help developers writing high quality unit tests*, *e.g.*, novel test code design patterns.

VII. Threats to validity

A number of factors could have influenced our findings.

Construct Validity. These threats refer to the research instruments used. Our confirmatory survey was conducted in a remote setting: as such, we could not verify the level of engagement of the participants or their behavior while working on the study. To tackle this issue, we did not include any mandatory questions other than discarding the incomplete submissions (more than 200 in total). To avoid any problem with the survey infrastructure, we ran preliminary tests amongst the authors as well as two external participants prior the public release of the survey. Similarly, we conduct a first pilot interview to consolidate and practice its structure. We advertised the survey on several social media platforms, in an attempt of reducing some selection bias. The majority of our participants engaged to the study via REDDIT, an independent forum that has been largely used in the past to ask for experts opinion about research topics [67].

External Validity. As for the generalizability of the results, in our study we first interviewed 5 expert with different background and level of experience. Our confirmatory survey collected answers from 70 participants with a diverse range of experience, employment, and team size, limiting possible threats to the validity of the given answer. Broader replication would be still be profitable to corroborate our findings. We selected 10 unit tests coming from the APACHE ecosystem. We tackle this threat by applying a random sampling of the tests while ensuring to not select more than one test from the same suite. In our future agenda we plan to both enlarge and diversify the application domain for the selected systems.

Conclusion Validity. With respect to the relation between treatment and outcome, the main threat is the selection of a wrong statistical model. To this aim, we verified the assumptions made by a proportional odds model [71]. In the first place, our dependent variable is naturally measured at an ordinal level. Secondly, we checked for multicollinearity amongst the covariants by exploiting hierarchical clustering based on the Spearman's rank correlation coefficient [63]. In particular, we discarded one variable for each pair having a $\rho > 0.6$. Finally, we check the assumption of proportional odds implying that the relationship between each pair of the outcome groups is the same. This assumes that the independent variables have the same effects on the odds regards the considered level. We test this assumption with the Brant test [9] implemented in the `brant R` package. We took some additional measures to avoid conclusion biases: in particular, we define the experience of the developer as control factor.

VIII. Conclusion and Future Work

In this paper, we investigated how developers perceive unit test code quality and how the metrics defined in literature match this perception. Through a mixed-method approach, featuring semi-structured interviews, a survey study, and a statistical modeling approach we discovered that existing metrics only partially match with the developer's perception of unit test quality, which should be complemented with (i) alternative/additional metrics to diagnose the actual usefulness of tests and (ii) automated documentation mechanisms that help developers understanding various aspects of test code, including assertions. These findings represent the main input for our future research agenda, which will be devoted to the development of novel metrics and mechanisms supporting developers with the assessment of test code quality.

REFERENCES

[1] Pizza versus pinsa: On the perception and measurability of unit test code quality — online appendix. https://anonymous.4open.science/r/1d53bcfe-348d-4a14-87d7-7a0dfec704d7/.

[2] K. Aggrawal, Y. Singh, and A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM, 2005.

[4] M. Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[5] 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. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012.

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

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

[8] D. Bowes, T. Hall, J. Petric, T. Shippey, and B. Turhan. How good are my tests? In *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 9–14. IEEE, 2017.

[9] R. Brant. Assessing proportionality in the proportional odds model for ordinal logistic regression. *Biometrics*, pages 1171–1178, 1990.

[10] R. P. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[11] 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.

[12] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci. How the experience of development teams relates to assertion density of test classes. In *ICSME*, pages 223–234. IEEE, 2019.

[13] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci. How the experience of development teams relates to assertion density of test classes. page to appear, 2019.

[14] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[15] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*.

[16] J. M. Converse and S. Presser. *Survey questions: Handcrafting the standardized questionnaire*. Number 63. Sage, 1986.

[17] J. W. Creswell. Mixed-method research: Introduction and application. In *Handbook of educational policy*, pages 455–472. Elsevier, 1999.

[18] M. Delahaye and L. Bousquet. Selecting a software engineering tool: lessons learnt from mutation analysis. *Software: Practice and Experience*, 45(7):875–891, 2015.

[19] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. Understanding flaky tests: The developer's perspective. page to appear, 2019.

[20] M. Ellims, J. Bridges, and D. C. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

[21] T. S. Flanigan, E. McFarlane, and S. Cook. Conducting survey research among physicians and other medical professionals: a review of current literature. In *Proceedings of the Survey Research Methods Section, American Statistical Association*, volume 1, pages 4136–47, 2008.

[22] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[23] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2011.

[24] M. Gambhir and V. Gupta. Recent automatic text summarization techniques: a survey. *Artificial Intelligence Review*, 47(1):1–66, 2017.

[25] M. Goeminne and T. Mens. Analyzing ecosystems for open source software developer communities. In *Software Ecosystems*. Edward Elgar Publishing, 2013.

[26] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82, 2014.

[27] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327, 2019.

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

[29] G. Grano, S. Scalabrino, R. Oliveto, and H. Gall. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th International Conference on Program Comprehension, ICPC*, 2018.

[30] S. Harispe, S. Ranwez, S. Janaqi, and J. Montmain. Semantic similarity from natural language and ontology analysis. *Synthesis Lectures on Human Language Technologies*, 8(1):1–254, 2015.

[31] C. A. R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[32] 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.

[33] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[34] H. Jin and F. Zeng. Research on the definition and model of software testing quality. In *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, pages 639–644. IEEE, 2011.

[35] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665.

[36] C. D. Kam, J. R. Wilking, and E. J. Zechmeister. Beyond the "narrow data base": Another convenience sample for experimental research. *Political Behavior*, 29(4):415–440, 2007.

[37] Y. W. Kim. Efficient use of code coverage in large-scale software development. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 145–155. IBM Press, 2003.

[38] P. S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 560–564. IEEE, 2015.

[39] G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *2006 17th International Symposium on Software Reliability Engineering*, pages 204–212, 2006.

[40] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel. How well do professional developers test with code coverage visualizations? an empirical study. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 53–60. IEEE, 2005.

[41] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[42] K. M. Lui and K. C. Chan. Pair programming productivity: Novice–novice vs. expert–expert. *International Journal of Human-computer studies*, 64(9):915–925, 2006.

[43] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[44] A. Marchenko, P. Abrahamsson, and T. Ihme. Long-term effects of test-driven development a case study. In *International Conference on Agile Processes and Extreme Programming in Software Engineering*, pages 13–22. Springer, 2009.

[45] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

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

[47] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In *16th IEEE international symposium on software reliability engineering (ISSRE'05)*, pages 10–pp. IEEE, 2005.

[48] R. O'Brien. A caution regarding rules of thumb for variance inflation factors, 10 2007.

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

[50] J. Offutt. A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098–1107, 2011.

[51] J. Pantiuchina, M. Lanza, and G. Bavota. Improving code: The (mis)

perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91. IEEE, 2018.

[52] D. L. Paulhus. Measurement and control of response bias. 1991.

[53] 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*.

[54] D. Qiu, B. Li, and H. Leung. Understanding the api usage in java. *Information and software technology*, 73:81–100, 2016.

[55] Y. Rafique and V. B. Mišić. The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, 39(6):835–856, 2012.

[56] J. M. Rojas and G. Fraser. Is search-based unit test generation research stuck in a local optimum? In *SBST@ICSE*, pages 51–52. IEEE, 2017.

[57] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427. IEEE, 2017.

[58] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018.

[59] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[60] J. Smith and H. Noble. Bias in research. *Evidence-based nursing*, 17(4):100–101, 2014.

[61] D. Spadini, F. Palomba, T. Baum, S. Hanenberg, M. Bruntink, and A. Bacchelli. Test-driven code review: an empirical study. In *Proceedings of the 41st International Conference on Software Engineering*, pages 1061–1072. IEEE Press, 2019.

[62] 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.

[63] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:88–103, 1904.

[64] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10–pp. IEEE, 2005.

[65] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15, 2016.

[66] 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.

[67] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020.

[68] J. L. Walker. Research column. the use of saturation in qualitative research. *Canadian Journal of Cardiovascular Nursing*, 22(2), 2012.

[69] Y. Wei, B. Meyer, and M. Oriol. Is branch coverage a good measure of testing effectiveness? In *Empirical Software Engineering and Verification*, pages 194–212. Springer, 2012.

[70] R. S. Weiss. *Learning from strangers: The art and method of qualitative interview studies*. Simon and Schuster, 1995.

[71] R. Williams. Generalized ordered logit/partial proportional odds models for ordinal dependent variables. *The Stata Journal*, 6(1):58–82, 2006.

[72] C. Winship and R. D. Mare. Regression models with ordinal variables. *American Sociological Review*, 1984.

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

[74] W. E. Wong, Y. Qi, L. Zhao, and K. Cai. Effective fault localization using code coverage. In *COMPSAC (1)*, pages 449–456. IEEE Computer Society, 2007.

[75] T. Yamaura. How to design practical test cases. *IEEE software*, 15(6):30–36, 1998.

[76] Y. Yu, G. Yin, T. Wang, C. Yang, and H. Wang. Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences*, 59(8):080104, 2016.

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

[78] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 214–224. ACM, 2015.