Noname manuscript No.
(will be inserted by the editor)

The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems

Fabiano Pecorelli · Fabio Palomba · Andrea De Lucia

Received: date / Accepted: date

Abstract Testing represents a crucial activity to ensure software quality. Recent studies have shown that test-related factors (e.g., code coverage) can be reliable predictors of software code quality, as measured by post-release defects. While these studies provided initial compelling evidence on the relation between tests and post-release defects, they considered different test-related factors separately: as a consequence, there is still a lack of knowledge of whether these factors are still good predictors when considering all together. In this paper, we propose a comprehensive case study on how test-related factors relate to production code quality in Apache systems. We first investigated how the presence of tests relates to post-release defects; then, we analyzed the role played by the test-related factors previously shown as significantly related to post-release defects. The key findings of the study show that, when controlling for other metrics (e.g., size of the production class), test-related factors have a limited connection to post-release defects.

Keywords Software Testing · Test Code Quality · Empirical Study

1 Introduction

Software testing is the activity that allows developers to check that the source code works as expected [79]. In the past, a number of researchers have investigated the properties that make test code more effective [9, 13, 33, 60, 61] as well as their relation to the ability of catching defects in production code [12, 13, 50]. Researchers have successfully demonstrated that the quality of test suites has a strong correlation with the post-release defects that appear in the production classes they test [13, 48], i.e., the higher the test quality the lower the likelihood that the corresponding production code will be affected

Fabiano Pecorelli, Fabio Palomba Andrea De Lucia
SeSa Lab - University of Salerno, Italy
E-mail: fpecorelli@unisa.it, fpalomba@unisa.it, adelucia@unisa.it

Listing 1 Example of test case considered in our study.

```
1  @Test
2  public void testAdd() {
3       SparseGradient x = SparseGradient.createVariable(0, 1.0);
4       SparseGradient y = SparseGradient.createVariable(1, 2.0);
5       SparseGradient z = SparseGradient.createVariable(2, 3.0);
6       SparseGradient xyz = x.add(y.add(z));
7
8       checkF0F1(xyz, x.getValue() + y.getValue()
9               + z.getValue(), 1.0, 1.0, 1.0);
10 }
```

by defects. For instance, Kochhar et al. [48] have shown that having a higher assertion density (measured as the number of assert statements per test lines of code) relates to a significantly lower number of defects in production code. Similarly, other studies have investigated the correlation between different types of code coverage and post-release defects metrics [9, 13] as well as test smells [90] on software quality, always reporting that test-related factors are relevant to explain the number of post-release defects in production code. It should be noted that by test-related factors we mean the set of metrics that can characterize the quality of tests, e.g., their design quality rather than their ability to cover the production code.

To provide the reader with a clearer understanding of how researchers in the past have studied the relation between the characteristics of tests and post-release defects, let consider the example reported in Listing 1.[1] It concerns with the test case named `testAdd`, which belongs to the `SparseGradientTest` test suite of the Apache Commons Math 3.3 system—one of the projects considered in our study. The test aims at verifying that the `add` method of the corresponding production class `SparseGradient` correctly sums a set of numbers; to this aim, it instantiates the variables to be added (lines #3, #4, and #5 of Listing 1) and sums them using the `add` method (line #6). Finally, it calls the method `checkF0F1`, implemented in the same test suite, that verifies the sum and checks for the first order derivative passed as additional parameters (line #9). The test case is able to entirely cover the corresponding production method (line coverage=100%) and, similarly, the entire test suite has a line coverage of 98%. In the subsequent release of Apache Commons Math, the class `SparseGradient` did not exhibit any defect: a possible reason lies in the ability of the corresponding test suite to provide developers with an effective instrument to verify the presence of defects and, as a matter of fact, previous work in literature have discovered a correlation between code coverage of tests and post-release defects in production code, i.e., the higher the coverage the lower the number of defects in subsequent releases of system [9, 13].

Despite the effort made by the research community in understanding the relations between test quality and post-release defects, we identify a key

[1] The suite to which the test case belongs has 1,128 lines of code - we report only an exemplary test case for the sake of understandability.

common limitation in previous work: they analyzed the impact of various test-related factors in isolation, controlling neither for other test-related factors nor for additional known phenomena affecting the quality (measured in terms of post-release defects) of production code (e.g., product metrics [3, 10]).

To clarify the practical effect of this common limitation, let consider again the example reported in Listing 1. While the code coverage was very high and suggested that the test suite could effectively help developers in spotting post-release defects, the fact that the class `SparseGradient` was actually defect-free in the subsequent release of Apache Commons Math might have and might not have been due to the high code coverage of `SparseGradientTest`. Other factors, for instance the low amount of maintenance activities performed on the production class, may have played a role. This is what actually happened to `SparseGradient`: it did not undergo any modification in Apache Commons Math 3.4 and, therefore, this was the reason making it defect-free—independently from the high value of code coverage of the corresponding test suite. Should this example be generalizable, it would mean that the findings reported in literature would not depict a clear picture on the relation between the characteristics of tests and their ability to foresee post-release defects.

In this article, we aim at addressing this limitation by proposing a case study on eight systems of the Apache ecosystem in which we explore how test-related factors identified in literature are related to software quality. As done in the previous studies, we operationalize software quality at software component level (in our case, file level) and by measuring the component's post-release defects. The main challenge of our work is represented by the extraction of a comprehensive set of test-related factors possibly influencing post-release defects. To address it, we first conduct a multivocal literature review on the test-related factors that have been associated to post-release defects in both white and gray literature, in an effort of eliciting a comprehensive set of metrics to consider in our study. Then, we build statistical models to study how the factors identified, i.e., (i) presence and executability of test suites, (ii) statically computable test code factors, and (iii) dynamically computable test code factors, relate to the number of post-release defects in production code, when also considering several product and process metrics. The main finding of our study is that most of the test-related factors do not have a direct relation with software quality, as opposed to factors such as production class LOCs and pre-release changes. Finally, dynamically computable test characteristics like code coverage have a relation to post-release defects, but only marginal. To sum up, this paper makes the following three main contributions:

1. A multivocal literature review on test-related factors known in white and gray literature to be related to post-release defects;
2. An empirical study investigating the role of the identified test-related factors on production code quality;
3. An publicly available online replication package, containing both dataset and scripts used in our analysis [78].

Structure of the paper. Section 2 discusses the related literature; in Section 3 we report the methodology and results of the multivocal literature review we conducted. Section 4 reports design and results achieved when studying the relation between test-related factors and software quality, while Section 5 discusses the main findings of the study, along with limitations and implications. Finally, Section 6 concludes the paper and highlights our future work.

2 Related Work

The main goal of our study is to assess the relationship between test-related factors and post-release defects. As such, in this section we discuss the research literature that in the past has proposed similar investigations.

Nagappan et al. [60] used the Software Testing and Reliability Early Warning (STREW-J) metric suite [60] to investigate the relation between in-process testing metrics and software quality. This suite includes a variety of test metrics belonging to three categories: (1) Test quantification, e.g., presence of test cases or assertion density, (2) Complexity and OO metrics, e.g., complexity and coupling of tests, and (3) size, i.e., the lines of code of tests. Their investigation—conducted on 54 small to large industrial companies—showed a significant relation between metrics in the suite and the emergence of post-release defects. These findings were later confirmed by Rafique and Misic [82], who pointed out that these metrics are even more effective in the context of test-driven development. With respect to these papers, ours aims to contextualize their results when considering a wider set of test-related factors known in literature to impact post-release defects. At the same time, we aim to shed lights on how much the power of test-related factors increases/reduces when additional factors related to production code are taken into account.

Other studies found a relation between test effort and product quality [61, 93] based on other testing metrics such as code coverage [9, 13, 61] and other static metrics (e.g., number of assertions) [60]. Kudrjavets et al. [50] showed the existence of a high correlation between assertion density and defect-proneness of production code, while Catolino et al. [12] showed that this relation may be due to the experience of the testing teams. In the experimental setting, these papers verified the relation of the considered test-related factors to post-release defects by considering the former alone, i.e., without controlling for possible confounding factors influencing the results. As such, the setting might lead to a limited view of the phenomenon: our paper addresses this limitation.

Chen and Wong [13] used code coverage for software failures prediction and showed that this metric influences code quality. Later, Cai and Lyu [9] confirmed this result. Nevertheless, a recent work by Kochhar et al. [48] contradicts those findings, reporting that coverage has an insignificant relation with the number of post-release defects. This cluster of papers shares the analysis methods employed: they relied on linear and logistic regression to understand how the considered test-related factors were correlated to the presence of defects

in future software releases. Also in this case, the test-related factors were considered alone and without additional confounding factors.

Spadini et al. [90] and Qusef et al. [81] studied the relation between test smells and software quality in terms of post-release defects. The former set the problem from a statistical perspective: test smells were controlled for the presence of code smells in production code as well as additional CK metrics computed on the exercised classes. While the key results of the study showed that smelly test suites make the production code more fault-prone, Spadini et al. [90] did not consider the effect of test smells when other test-related factors are included. The latter first analyzed the evolution of test smells in Apache Ant; then, they used correlation analysis to study the relation between test smells and post-release defects, finding a positive correlation. This paper shares the same limitations of the other previous works, hence not considering neither other test-related nor confounding factors - which is the object of our study.

To sum up, the papers discussed above report on the effectiveness of tests to reveal the likelihood of post-release defects. As such, they represented the ground based on which we built the empirical study presented herein. Nonetheless, we aim at making a further step ahead toward the understanding of the relation between test-related factors and software quality: as discussed in the remainder of the paper, we considered the metrics used by previous work as independent variables of our empirical study.

As a final note, the first author of this paper recently published an extended abstract [76] reporting some preliminary results of this work on three software systems, which are extensively confirmed in our journal paper. Indeed, unlike our new contribution, the previous publication (1) did not systematically gather test-related factors, i.e., the multivocal literature review was not part of the extended abstract, (2) did not consider many of the confounding factors that may potentially influence the results (e.g., code smells), (3) did not perform a three-level analysis, i.e., impact of presence/executability, of static and dynamic factors, (4) did not analyze the achieved findings in detail, and (5) did not provide insights and discussion about the potential impact that this work has for further research opportunities.

3 Collecting Test-Related Factors: A Multivocal Literature Review

A critical challenge in this study concerns with the definition of a comprehensive set of test-related factors to experiment with. We approached this challenge by conducting a Multivocal Literature Review (MLR) [31] to identify the test-related factors that might influence the quality of the exercised production code. An MLR is an enhanced version of systematic literature reviews that not only considers *white papers*, i.e., those that have been published in conferences and journals, but also *gray documentation*, i.e., the knowledge that can be extracted from online unpublished sources like websites and blog posts. Next, we describe methodology and results achieved from the literature review.

3.1 Research Methodology

The *goal* of the multivocal literature review is to collect the test-related factors that have been analyzed and/or discussed in both previously published work and online unpublished sources, with the *purpose* of providing a comprehensive view of which factors have been associated to post-release defects. The *perspective* is that of researchers who are interested in gaining knowledge of test-related factors and their relation with software quality.

3.1.1 Research Question

To address the goal of our study, we set up the following research question:

> **RQ_0.** *What are the test-related factors related to post-release defects, according to the available white and gray literature?*

As further reported in this section, we followed well-established research guidelines to conduct systematic and multivocal literature reviews [31, 46].

3.1.2 Search Query Definition

The search query represents the set of keywords that are used to search reliable sources on the phenomenon of interest [46]. In our case, we made two main considerations before defining it. First, we noticed that multiple terms could be used as synonym of *'defect'*: these are *'bug'*, *'fault'*, and *'failure'*.[2] Secondly, the term *'post-release'* could also be referred to in different ways, namely *'post-production'*, *'post-delivery'*, and *'post-verification'*. According to these considerations, we defined the following search query:

(*'test'*) **AND** (*'post-release'* **OR** *'post-production'* **OR** *'post-delivery'* **OR** *'post-verification'*) **AND** (*'defect'* **OR** *'bug'* **OR** *'failure'* **OR** *'fault'*)

3.1.3 Selecting the Source Engines

The selection of relevant sources is a crucial activity to provide a comprehensive description of the state of the art [31, 46]. In our context, this step consisted of selecting search engines that could cover both white and gray literature. As for the former, we selected all major databases indexing published papers: these are (1) the IEEEXplore Digital Library,[3] (2) the ACM Digital Library,[4] (3) Science Direct,[5] (4) SpringerLink,[6] and (5) Scopus.[7]

[2] We did not include the term *'error'* since it refers to the *action* performed by a developer to introduce a defect in source code rather than to the defect itself [79].

[3] Link: https://ieeexplore.ieee.org/Xplore/home.jsp

[4] Link: https://dl.acm.org

[5] Link: http://www.sciencedirect.com

[6] Link: https://link.springer.com

[7] Link: https://www.scopus.com

The selection of these search engines was driven by our willingness to consider as many sources as possible when conducting our literature search. These databases are widely recognized as the most representative for research in the field of software engineering [8, 42] and contain a massive amount of resources, i.e., journal articles, conference and workshop proceedings, books, etc., concerned with the research question we posed.

As for the gray literature, we followed a similar approach as other multivocal literature reviews (e.g., [30, 39]) and exploited the Google search engine.[8]

3.1.4 Exclusion and Inclusion Criteria Definition

Exclusion and inclusion criteria report the characteristics that a retrieved source must not (or must) have to be considered useful for addressing the research question [31, 46]. Also in this case, we needed to define criteria depending on whether a resource comes from the white or the gray literature, as some characteristics might not be applied for gray resources.

As for the white literature, we adopted the following exclusion criteria:

- Articles that were not focused on investigating the relation between test-related factors and post-release defects, e.g., papers studying how test smells relate to mutation coverage;
- Articles that have later been extended; particularly, in case of a conference paper has been extended to journal, we only considered the journal article as it is more complete.
- Articles not reporting any empirical validation of the relation between test-related factors and post-release defects, e.g., non-validated conjectures of the existence of a relation between test smells and code coverage;
- Articles that were not written in English;
- Articles whose full text was not available;
- Articles that did not undergo a peer-review process, e.g., M.Sc thesis;
- Duplicate papers retrieved by multiple databases.

We set one main inclusion criterion:

- Articles reporting an empirical validation of the relation between test-related factors and post-release defects, e.g., papers studying how test smells relate to post-release defects.

It is worth noting that we did not set any temporal limit to our search, as we were interested in retrieving all possible sources for conducting a comprehensive analysis of test-related factors and post-release defects.

Turning the attention to the gray literature, the main challenge was represented by the assessment of the reliability of a source. Indeed, among the resources retrieved, there might be some that did not have the minimum quality

[8] Link: https://www.google.com

to be considered reliable for our literature review. To take this aspect into account, we assessed the reliability of gray resources by relying on the guidelines provided by the University of Wisconsin[9] and, according to them, excluded unreliable resources. In particular, these guidelines are:

- *Author*. Information on the internet with a listed author is an indication of a credible site. If an author is willing to stand behind the information presented (and in some cases, include his or her contact information) is a good indication that the information is reliable.
- *Date*. The date of any research information is important, including information found on the Internet. By including a date, the website allows readers to make decisions about whether that information is recent enough for their purposes.
- *Sources*. Credible websites, like books and scholarly articles, should cite the source of the information presented.
- *Domain*. Some domains such as .com, .org, and .net can be purchased and used by any individual. However, the domain .edu is reserved for colleges and universities, while .gov denotes a government website. These two are usually credible sources for information. Websites using the domain .org usually refer to non-profit organizations which may have an agenda of persuasion rather than education.
- *Site Design*. This can be very subjective, but a well-designed site can be an indication of more reliable information. Good design helps make information more easily accessible.
- *Writing Style*. Poor spelling and grammar are an indication that the site may not be credible. In an effort to make the information presented easy to understand, credible sites watch writing style closely.

Of course, we only considered resources written in English and that were fully available for reading. At the same time, to include a resource in our study we defined the following two criteria:

- The resource must report on practitioner's experiences and/or discussion of using test-related factors to establish the likelihood to have defects in production code;
- The resource must describe the test-related factor(s) it refers to, i.e., it must clearly mention that the test executability represents an important factor to assess post-release defects.

3.1.5 Execution of the Multivocal Literature Review

Once defined the ground for our multivocal literature review, we proceeded with its execution. Figure 1 overviews the process, reporting the inputs/outputs of

[9] Link: https://uknowit.uwgb.edu/page.php?id=30276

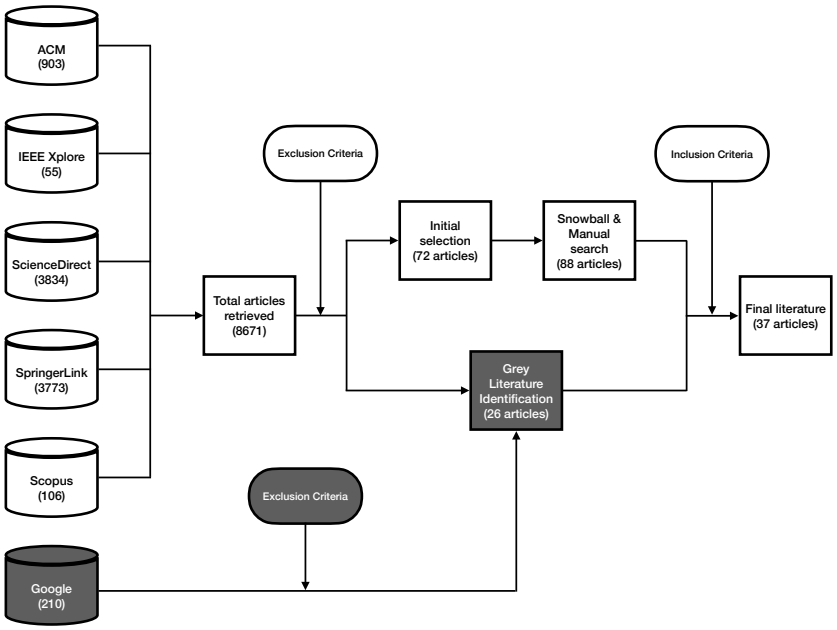

Figure 1 Steps performed when conducting the multivocal literature review.

each stage as well as summarizing the number of resources retrieved from each search engine and considered for our study. For the sake of understandability, the figure reports in white background the steps referring to the white literature review, while in gray the parts related to the gray literature.

The entire process was jointly executed by the first two authors of the paper in the period between May 11 to June 10, 2020. Whenever possible, the two authors met physically to perform the tasks; otherwise, they conducted the review through Skype. The entire execution took around 80 person/hour: the inspectors started analyzing the white literature, as this was supposed to take longer. Upon completion, they then focused on the gray literature.

As shown, when executing the search query on the white literature search engines, we obtained a total of 8,671 hits: most of them came from Science Direct and SpringerLink, which returned 3,834 and 3,773 results, respectively. The other databases were instead more restrictive when returning results.

The inspectors applied the exclusion criteria on the initial set of papers retrieved. In so doing, they mainly focused on title and abstract; nevertheless, in cases where the exclusion criteria could not be applied solely looking at these pieces of information, they explored the content of the paper more, for instance by reading the research questions, methodology, or conclusion of the paper. The exclusion criteria led to the removal of the vast majority of the initial sources (8,599 papers), giving as output a candidate set of 72 papers.

At this point, the inspectors adopted a backward and forward snowballing approach as recommended by Wohlin [102]. This was based on one iteration which consists of examining (i) the outcoming citations, namely the list of papers cited by the article under investigation and (ii) the incoming citations, namely the list of papers citing the article under investigation, with the aim of augmenting the candidate set of papers with additional resources that were not identified through the search engines. In the first case, the inspectors went over the citations reported in the candidate papers. In the second case, they used Google Scholar[10] to retrieve the list of papers citing the candidate ones. As previously done, the inspectors first focused on the title of a cited/citing paper: if it was not possible to establish the actual relevance of a new resource, they downloaded it and started reading its content. At the end of this stage, the inspectors included 16 papers to the candidate set, which reached 88 sources.

Finally, the inclusion criterion was applied: 11 papers passed the examination, forming the final set of white resources to be considered in our study.

As for the gray literature review, the inspectors followed a similar methodology. When executing the search query on Google, it returned a total of 210 results, distributed on 21 pages. It is worth noting that the inspectors performed the entire gray literature identification process using the 'incognito' mode to avoid that their personal navigation history could bias the results of the search process. Given the limited amount of results, the inspectors could browse each of them and apply the specific exclusion criteria established for the gray literature search. The joint work allowed them to immediately discuss the suitability and reliability of the resources analyzed: as an outcome, they identified a set of 26 candidate resources. Interestingly, all of them passed the inclusion criteria, composing the final set of gray resources and contributing to the grand total of 37 sources included in our multivocal literature review.

3.1.6 Quality Assessment and Data Extraction Process

As a final step of our multivocal literature search, we assessed the quality of the retrieved literature sources and extract data about the test-related factors associated with post-release defects.

In particular, after the selection process of both white and gray literature, we defined a checklist to assess the reliability and thoroughness of the selected sources. The checklist included the following questions, which have been defined with the goal of determining whether the considered sources actually treated test-related factors and their relation with software quality:

1. Are the test-related factors mentioned in the source clearly defined?
2. When considering white papers, are the test-related factors actually assessed against post-release defects?
3. When considering white papers, is the research methodology clearly stated?

[10] Link: https://scholar.google.com

4. Are the conclusions stated supported by data (for white papers) or clear motivations (for gray sources)?

To each of these questions, the inspectors could reply with *'Yes'*, *'No'*, *'Partially'*. The inspectors considered a study as partial in cases where the methodological details could have been derived from the text, even if they were not clearly reported. These answers were scored as follows: *'Yes'*=1, *'Partially'*=0.5, and *'No'*=0. For each primary study, its quality score was computed by summing up the scores of the answers to all the four questions. At the end of the process, the inspectors classified the quality level into *High* (score = 4 for white papers, score = 3 for gray articles since they did not include point 3 of the quality assessment), *Medium* ($2 \leq$ score < 4 for white papers, score = 2 for gray literature), and *Low* (score = 1). According to our assessment, 2 resources were classified as *Medium* and 34 as *High*. Therefore, we could be able to extract data from all the selected sources.

As for the actual extraction process, the inspectors proceeded as follow. For white papers, they looked at the experimental design in order to identify (1) the test-related metrics used as independent variables and (2) the dependent variable adopted (i.e., post-release defects). Hence, in this case the data extraction was only based on these two elements. As for gray sources, the inspectors looked at the entire text to interpret the practitioner's opinions and elicit the test-related factors they were referring to. For example, let consider the following case, which comes from the source [S03] in the Appendix A:

"Under the assumption that tests are of good quality, [code coverage] can uncover which parts of the software have a known level of defects vs. unknown."

As reported, in the text above the practitioner refers to code coverage and how it can be used as an indicator for the location of faults in production code.

3.2 Analysis of the Results

This section summarizes the results of our multivocal literature review. It is worth remarking that we report the entire list of sources considered when performing the review in Appendix A.

Figure 2 depicts the distribution over years of the resources retrieved in our review. There are two key observations to report by looking at the figure. In the first place, we observe that the number of papers published on this topic is particularly low (11) and most of them are rather recent: indeed, in the last three years we observe a slightly increasing trend. In the second place, it is surprising to see that the number of gray resources is higher than the one of white literature papers: this aspect seems to suggest that the problem of assessing the power of test-related factors to forecast post-release defects has been neglected by the research community, while it represents something important for practitioners. Also in this case, we notice that the number of gray articles increases in the last few years. Intuitively, this aspect may be due

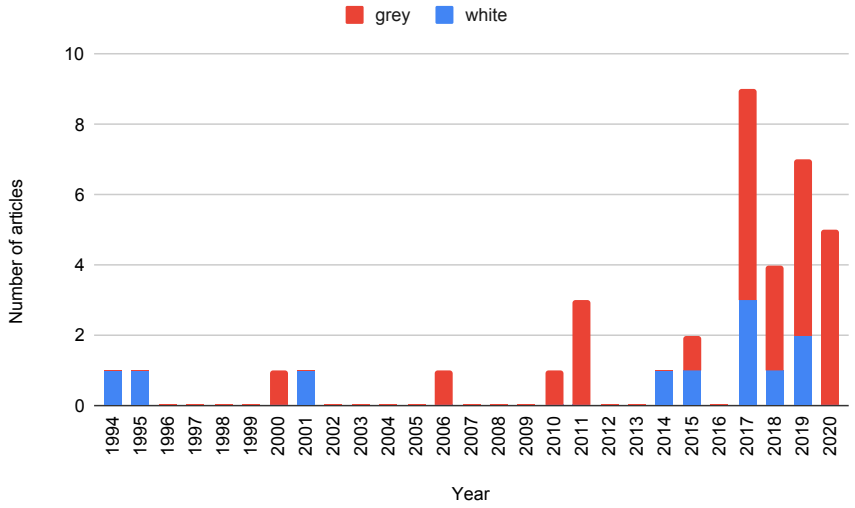

Figure 2 Total number of articles retrieved over years. Red bars refer to the gray resources, blue ones to white papers.

to the steady raise of development models that require the frequent execution of tests (e.g., continuous integration) and that somehow enforce developers in keeping the characteristics of tests into consideration, hence creating a growing interest into the relation between tests and software quality.

Turning the attention to the types of test-related factors mentioned in the retrieved resources, Figure 3 overviews the themes extracted by analyzing each of them. Interestingly, most of the gray articles mentioned *presence and executability* of test classes: in particular, a number of practitioners point out that having a properly set environment represents a crucial factor that enables the prompt identification of defects in source code. As an example, in the resource [S25], the Chief Technology Officer of the Cockroach Labs—a well-known company that develops relational databases for cloud-native applications—reports on a two-year experience with using an open-source framework for testing distributed databases, i.e., Jepsen.[11] He explains that an effective method to make tests actionable is to have a strong environment and, indeed, quoting from [S25]: *"every night we start up a 5-node cluster and run each test+nemesis combination for 6 minutes each"*.

Another aspect deemed as important by most practitioners is represented by *dynamic factors*, e.g., code coverage. We can notice that this type of factors has attracted most of the attention of the research community, which frequently investigated how these factors can influence post-release defects. At the same time, from the gray literature emerged the value of *static factors*, e.g., test code

[11] Link: https://jepsen.io

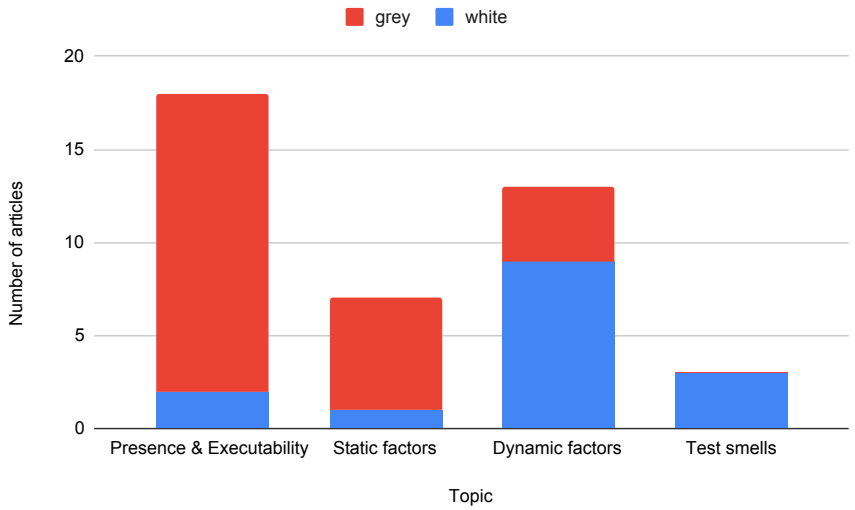

Figure 3 Total number of retrieved articles divided per theme. Red bars refer to the gray resources, blue ones to white papers.

metrics—an aspect that seems to be overlooked by the research community. For instance, in the resource [S16], the founder of SoftwareTestingMaterial—a blog reporting and discussing on testing practices and methodologies—reports that test code metrics can *"monitor and control process and product. [They] help to drive the project towards our planned goals without deviation"*.

Finally, the last type of test-related factor emerging from our multivocal literature review concerns with *test smells*, namely sub-optimal design or implementation solutions applied when developing test code [55]. This aspect was, however, only mentioned and investigated by researchers in the past, while we did not observe gray literature reporting on it. This possibly corroborates previous findings in the field reporting that practitioners do not perceive test smells as actual problems [94].

To conclude the discussion of the results for the multivocal literature review, Table 1 reports the specific metrics identified for each category of test-related factors. As shown, we identified 14 test-related factors. First, we extracted metrics related to presence and executability of test classes, which are connected to the testing environment. Secondly, we identified structural metrics [14] such as test lines of code (LOC), test complexity (WMC), test coupling (EC), and assertion density. Also, we found classical metrics like line and mutation coverage [2]. Finally, we found five test smell types, i.e., *Assertion Roulette*, *Eager Test*, *Indirect Testing*, *Resource Optimism*, and *Mystery Guest*: these were the test smells associated to defect-proneness in previous work [81, 90].

Table 1 Test-related factors resulting from the multivocal literature review.

Group	Name	Description
Presence and Executability	Availability of test classes	The availability of a test suite for a production class.
	Executability of test classes	The ability to run a test case for a given production class.
Static factors	TLOC	Number of lines of code of the Test Suite.
	TWMC	Weighted Method Count of the Test Suite.
	TEC	Efferent coupling of the Test Suite.
	Assertion Density	Percentage of assertion statements in the test code (i.e., number of assertions / T_LOC).
Test smells	Assertion Roulette	A test containing several assertions with no explanation.
	Eager Test	A test case testing more methods of the production target.
	Indirect Testing	A test interacting with the target via another object.
	Resource Optimism	A test that make optimistic assumptions on the existence of external resources.
	Mystery Guest	A test that use external resources (e.g., files or databases).
Dynamic factors	Line Coverage	Percentage of statement in production class that are covered by the test.
	Branch Coverage	Percentage of branches in production class that are covered by the test.
	Mutation Coverage	Percentage of mutated statement in production class that are covered by the test.

RQ0 - Findings

From the multivocal literature review, we discovered four categories of test-related factors that have been associated to post-release defects in white and/or gray literature. These are connected to presence and executability of tests, static and dynamic test code metrics, and test smells.

4 Studying the Relation between Test-Related Factors and Post-Release Defects

Once collected the set of test-related factors to investigate, we proceeded with the definition of an empirical study to verify their relation with post-release defects. In the following sections, we describe the methodological choices done as well as the results achieved.

4.1 Research Methodology

The *goal* of the empirical study is to investigate how test-related factors are related to post-release defects, with the *purpose* of measuring the explanatory

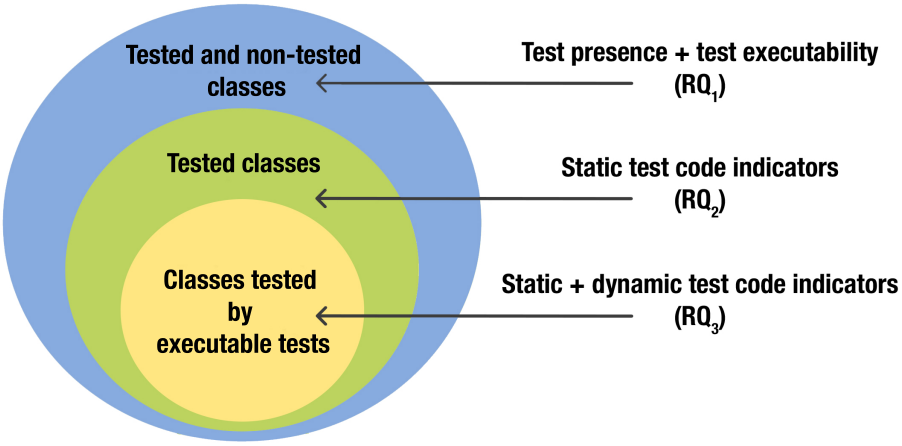

Figure 4 The three levels of investigation considered in our empirical study, by research question.

power of having a good test suite on software code quality. The *perspective* is of both researchers and practitioners: the former are interested in assessing the relation between test-related factors and post-release defects, while the latter are interested in understanding the extent to which good quality test suites can help them to reduce post-release defects.

4.1.1 Research Questions and Methodological Sketch

Our study is structured around three main research questions. Figure 4 synthesizes the levels of our analyses. In an ideal scenario, all production classes in a software project have corresponding tests that can be successfully executed. However, in practice this is rarely the case, which represents an interesting scenario for us, because it creates a *natural experiment* where we can compare software quality (measured as post-release defects) in both tested and untested classes. Therefore, we started our empirical study by investigating the role played by the presence of test classes and their executability with respect to software quality.

> **RQ$_1$.** *How do presence and/or executability of test classes relate to post-release defects of production code?*

We further investigated the role of testing on source code quality at a finer level of granularity. We restricted our analysis to those test-related factors that can be statically computed (e.g., test size)

> **RQ$_2$.** *How do statically computable test code factors relate to post-release defects of production code?*

The final part of our empirical study was focused on the understanding of how dynamic factors (e.g., code coverage [2]) relate to software quality. This maps a scenario in which production classes are exercised by executable tests.

> **RQ$_3$.** *How do dynamically computable test code factors relate to post-release defects of production code?*

From an experimental design perspective, for each research question we established a set of *subjects*, *factors*, and *treatments* - as also illustrated in Figure 4. As for **RQ**$_1$, we considered all the classes of the considered systems as *subjects*, taking into account the presence and executability of tests as *factors*, with the aim of assessing the extent to which they affect source code quality (i.e., the *treatment*). In **RQ**$_2$, we narrowed the *subjects* of the study so that we could only consider the classes having at least one test associated to compute *factors* computable using static analysis (i.e., metrics and code quality indicators) and assess their impact on post-release defects. Finally, in **RQ**$_3$, the *subjects* are represented by the set of classes having at least one executable test, while the *factors* are the dynamically computable metrics: these are used to establish a relation between dynamic test factors and software quality. The following sections report on the subject dataset as well as the steps conducted to answer our **RQ**s.

4.1.2 Context selection

The *context* of the study consisted of eight open-source software systems, whose characteristics are shown in Table 2. Their selection was driven by three main requirements of both our tooling and data analysis procedures. In the first place, we focused on Java because most of the test-related factors experimented can be only computed for this programming language (e.g., test smells are only defined and validated by our community for Java [97]). Secondly, we restricted our analyses to systems having a large change history information, as it is our willingness to have projects with (i) a number of post-release defects to allow significant analyses—we excluded systems with less than 50 defects reported in the corresponding issue tracker[12]—and (ii) a number of previous changes—we excluded systems with less than 500 commits—that can be used to control for our results, as detailed in Section 4.1.5. Finally, we required the selected systems to have at least one compilable release to use for our study [95], as some of the considered test-related factors can only work with compilable source code (e.g., dynamic information like line and mutation coverage). Moreover, to allow the application of the state-of-the-art tool for line and mutation coverage (i.e., PiTest), we checked for projects having no sub-modules and built with Maven, using the default Maven directory structure (i.e., the production classes are located in the 'src/main' package, while the test classes are in the 'src/test'

[12] We retrieve the link to the issue tracker adopted by a certain project through the analysis of the developer's contribution guidelines.

package). As a result of this selection procedure, we found a family of eight projects that met all the aforementioned requirements, i.e., APACHE-COMMONS. In our online appendix [78] we provided an Excel sheet where we detailed the context selection and filtering procedure.

In an effort of providing more details about the context of our study, as recommended by a recent work [6], we manually dived into the information reported in the repositories analyzed, particularly looking at (1) contribution guidelines, which may indicate some relevant information of how contributors are supposed to perform testing as well as on the development process in place and (2) the change history, which reports data related to frequency of releases, number of contributors, and so on. In cases where contextual information were not minable looking at the above mentioned sources (e.g., testing type), we also analyzed online documentation (e.g., guidelines of the APACHE SOFTWARE FOUNDATION) or directly looked at the source code and extracted the remaining data. As a result, we discovered that the eight selected projects share similar characteristics as well as a similar development community—as somehow expected, since they belong to the same family of projects. APACHE COMMONS is a *commit-then-review* community, so developers who want to contribute should follow APACHE's code of conduct[13] and announce their intentions and plans on the developers mailing list before committing code. Releasing a new version of the system requires the vote of a PMC (Project Management Committee) according to the APACHE Release Creation Process.[14] For this reason, new code is not released at regular time intervals but its release depends on several aspects. Tests are written and committed together with production code indicating that developers adopt a *test-as-you-write* development strategy. All the tests available in the considered systems are written at unit-level, following a black box strategy.

Table 2 Systems from Apache Commons considered in the study

Name	# Commits	# Releases	# Contributors	# Production Classes	# Test Classes	# Defects
Codec	1,792	45	39	26	31	134
Collections	3,091	49	51	270	139	341
DBCP	1,983	62	34	53	21	367
DbUtils	656	29	21	25	20	56
IO	5,400	54	58	100	47	281
Lang	2,141	89	140	119	98	634
Math	6,395	65	34	804	404	684
Pool	1,879	168	38	41	14	205

4.1.3 Dependent Variable

The dependent variable of our study is the number of post-release defects. To compute it, we first determined whether a commit fixed a defect. This was

[13] `http://www.apache.org/foundation/policies/conduct.html`

[14] `https://infra.apache.org/release-publishing.html`

done by employing the textual-based approach proposed by Fischer et al. [27], which is based on the analysis of commit messages. Such an approach has been extensively used in the past [41, 45] and was assessed to have an accuracy close to 80% [27, 71]. Specifically, we searched for issue IDs in commit messages by finding matches with the prefix used in the bug tracker system. Once retrieved a commit referencing an issue, we queried the apache issue tracker system's APIs in order to filter only issues related to resolved bugs. Then, we searched for keywords indicating fixing activities in the commit message, such as *'bug'*, *'fix'*, or *'defect'*, in order to select only the bug-fixing commits. Once we detect all bug fixing commits, we employed the SZZ algorithm [88] to obtain the commits where the defect was introduced. In particular, the SZZ algorithm relies on the annotation/blame feature of versioning systems [88]: given a defect-fix activity identified by the defect ID k, the approach works as follows:

- For each file f_i, $i = 1 \dots m_k$ involved in a defect-fix k (m_k is the number of files changed in the defect-fix k) and fixed in its revision *rel-fix*$_{i,k}$, we extracted the file revision just *before* the defect fixing (*rel-fix*$_{i,k} - 1$).
- Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the defect k, we identified the production class C_j to which the changed line belongs. Furthermore, the `blame` feature of `Git` is used to identify the revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified and excluded using an island grammar parser [57]. This produces, for each production class C_j, a set of $n_{i,k}$ defect-inducing revisions *rel-defect*$_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

Once retrieved the list of defect-inducing commits, we computed post-release defects of a class as the number of defect-inducing activities involving the class in the period after the selected release r_j. To compute the dependent variable, we use the PyDriller framework [89], which implements the SZZ algorithm. Some recent work has reported that the SZZ algorithm has a low accuracy, in particular concerning precision [18, 84]; should these observations be verified on our dataset, this would threat the validity of our results. For this reason, we manually validated the performance of the SZZ algorithm on our dataset. Specifically, the first two authors of this paper (the *inspectors*) jointly analyzed all the 251 total defect-inducing commits output by SZZ. In doing so, they relied on the source code of both defect-fixing and defect-inducing versions of the projects: the task was performed to understand whether a change in the defect-inducing version actually introduced the defect that was fixed in the defect-fixing version. The inspection pointed out a precision of 92% (231 correct defect-inducing commits): this result diverges from previous findings [84] and suggests that the performance of the algorithm is strongly dependent on the considered projects. As a consequence of this validation, we excluded the 20 false positives from the analysis to have a more accurate dataset.

4.1.4 Independent Variables

We defined a different set of independent variables for the three **RQ**s.

Independent variables for RQ_1. In the first research question, we defined two independent variables. The first one was represented by the presence of a test class for each production class of the selected systems. We defined a variable named *'is-tested'* that assumes the value *'true'* if the production class has a test class that exercise it, *'false'* otherwise. With respect to the selection of this independent variable, there are two observations to be done. Intuitively, the presence of test classes alone cannot affect software quality—having a test does not imply the identification of defects. However, having them is a *necessary* condition: the conjecture behind the selection of *'is-tested'* as independent variable was that the availability of a test suite may allow developers to promptly identify defects, hence affecting the number of post-production defects. Hence, we conjectured an indirect relation and aimed at verifying it. In this respect, it is also worth noting that the presence of tests was one of the factors mentioned by practitioners when analyzing the gray literature: this means that the availability is actually perceived as a relevant factor for software quality. This supports the idea of an indirect relation of this factor with post-release defects.

A key point for the computation of the independent variable was related to linking each test class to each of the considered production class. All the selected projects rely on Maven as build tool. Thus, to perform the linking, we relied on the `pom` file, which contains the rules to identify the test classes to execute when the projects need to be built or packaged. In particular, we first identified all production and test classes by scanning the `pom` file and looking for the `sourceDirectory` and `testSourceDirectory` fields, that indicate the location of production and test code, respectively. When the fields were not reported explicitly, we considered the default source and test directories. Afterwards, we used a pattern matching approach based on naming conventions to find the production class related to a certain test class, as it has been done in previous work [32, 52, 94]: given the name of a production class (e.g., '`ClassName`') belonging to the `sourceDirectory` folder, it checks for the presence of a test class having the same name as the production class but with the prefix or postfix "Test" in the `testSourceDirectory` (e.g., '`ClassNameTest`' or '`TestClassName`'). In case the approach cannot identify a test for a certain class, the variable *'is-tested'* for the considered production class is "false", "true" otherwise. The accuracy of this linking approach has been previously assessed [98]: it showed an accuracy close to 85% and is comparable with more sophisticated (but less scalable) techniques (e.g., slicing-based approaches [80]).

The second independent variable is named *'are-tests-executable'*: it assumes the value *"true"* if the tests exercising the production class can be ran, *"false"* otherwise. Also in this case, the selection was supported by the results achieved when analyzing the gray literature: indeed, multiple times practitioners reported that having tests that can be actually run against the production code is an important factor to assess post-release defects. In this sense, we aim at providing

evidence of the effect of having runnable tests on software quality. To determine the value of the variable, for each considered project we ran the `mvn verify` command, which executes all the available tests. If the execution of a test proceeds without errors,[15] then we considered the test as executable. Otherwise we marked it as non-executable.

With respect to the executability of tests, it is worth noting that we considered the building environment directly used by developers of the considered Apache projects. In other words, the methodology we used to run tests is exactly the same as the one used by the actual developers. Hence, we considered a real-case scenario of use of the tests, i.e., we considered a *natural experiment* that considers what actually happens in practice.

Independent variables for RQ_2. In the second research question, we studied the relation between static test-related factor and post-release defects. So, we considered test-related factors that can be computed without executing test classes: these are test code metrics and smells (see Table 1). To compute the former, we relied on the tool by Spinellis [92]. As for the latter, we used the code-metrics based tool developed by Bavota et al. [4]. The detector is able to identify instances of five test smell types, namely *Mystery Guest*, *Resource Optimism*, *Eager Test*, *Assertion Roulette*, and *Indirect Testing*. Detailed definitions of the smells are given in our online appendix [78]. All the considered smells have been related to defect-proneness by previous work in the field [81, 90]. The selection of the test smell detector was driven by the high accuracy it showed in previous studies, with F-Measure close to 86% [4, 66, 72].

Independent variables for RQ_3. In the context of the third research question, we also included the metrics that can be computed only when executing the test classes: the dynamic factors (i.e., Line Coverage, Branch Coverage, and Mutation Coverage). As for line and branch coverage, we used Cobertura.[16] For mutation coverage, we used PiTest.[17] It is worth noting that the choice of using PiTest was not only driven by the fact that this is the state-of-the-art tool for mutation testing [32, 51], but also by the relatively low number of equivalent mutants, i.e., mutants having a semantically similar behavior [1], it generates: indeed, a recent study by Fernandes et al. [26] reported that only 20% of the mutants generated by PiTest can be considered equivalent, which is substantially lower than other mutation testing tools available in literature.

4.1.5 Confounding Factors

In addition to the test-related factors we collected from our MLR (see section 3), the number of post-release defects may be due to other factors related to the structure of production code [3]. Thus, to avoid a biased interpretation of the

[15] Note that a Maven error explicitly indicates that the test cannot be run for some reasons, as opposed to a failure, which instead reports that the test has found an anomalous behavior in the production code.

[16] Link:https://cobertura.github.io/cobertura/

[17] Link: http://pitest.org

results, we introduced a set of well-known source code and process metrics as confounding factors, which are summarized in Table 3. All of them have been previously related to defect-proneness, as further explained in the following:

– We considered the **Lines Of Code (PLOC)** metric, that measures the size of production classes. According to previous findings [49, 67, 104], the larger a class the higher its fault-proneness. As such, the number of post-release defects might be a reflection of the production code size and, therefore, we computed LOC to control our findings on the impact of the presence of test suites. To measure PLOC, we used the tool devised by Spinellis [92].

Table 3 List of confounding factors used in the study.

Group	Name	Description
Static factors	PLOC	Number of lines of code of the Production Class
	PWMC	Weighted Method Count of the Production Class
	PEC	Efferent coupling of the Production class
Code smells	God Class	A class having a large size, poor cohesion, and several dependencies with other data classes of the system
	Class Data Should Be Private	A class exposing its attributes, thus violating the information hiding principle
	Complex Class	A class presenting a overly high cyclomatic complexity
	Functional Decomposition	A class implemented as a function
	Spaghetti Code	A class that exhibit a functional-style programming structure, declaring a number of long methods without parameters
Process Metrics	Pre-release Changes	Number of changes involving the Production class before the release date of the considered snapshot

– We computed **Weighted Method per Class (PWMC)** [14] as a way to measure the complexity of production code. A number of previous studies has shown the metric to be related to the number of defects in which a production class will incur [22, 62, 105]. The tool by Spinellis [92] was used to compute the metric on our dataset.
– We measured the **Efferent Coupling (PEC)** of production classes because, as reported by previous research [3, 19, 29, 47, 87], the higher the coupling of a class the higher its fault-proneness. Also in this case, we employed the tool by Spinellis [92] to compute PEC.
– We considered **code smells**, i.e., symptoms of the presence of poor implementation choices [7, 28], since they are reported to be connected to the fault-proneness of production code [20, 36, 43, 68, 69, 70, 77, 96]. We considered five code smells from the catalog by Fowler [28] that have different characteristics, namely *God Class*, *Class Data Should Be Private*,

Complex Class, *Functional Decomposition*, and *Spaghetti Code*. We provided a complete definition of those smells in our online appendix [78]. These code smells have been analyzed by previous work studying their effect on source code defect-proneness [43, 70]. Therefore, our selection was driven by these findings. As for the actual detection of these code smells, we relied on DECOR [56], a state-of-the-art detection tool which has shown an accuracy close to 80% [56]. In our work we re-evaluated the precision of DECOR. The two authors previously involved in the validation of the test smells also conducted this analysis: they manually validated all the 137 code smell instances output by the tool. The task was to understand whether a certain code smell candidate given by DECOR actually revealed the existence of a design problem in source code. After the first assessment, the two inspectors compared their evaluations, reaching an agreement of 95%. The remaining 5% of cases (i.e., seven code smell candidates) were discussed and, finally, four of them turned to be real code smells. Following this validation, we (i) confirmed the good accuracy and the suitability of DECOR in our context and (ii) excluded the false positive smells from our analysis.

– We computed the number of **pre-release changes** and **pre-release defects** because metrics capturing the previous history of production classes can reveal relevant evolution aspects [37, 83]. To compute the number of pre-release changes, we mined the change log of the considered projects and count how many times a certain production class has been modified. As for the pre-release defects, we relied again on the SZZ algorithm implemented in PYDRILLER [89].

4.1.6 Statistical Modeling and Data Analysis

After collecting the data for all the considered projects, we defined three groups of hypothesis, related to the three research questions.
As for $\mathbf{RQ}_1$, we defined two pairs of null (Hn) and alternative (An) hypotheses:

Hn1. There is no correlation between the presence of test classes and software quality, as measured by post-release defects.

An1. There is a correlation between the presence of test classes and software quality, as measured by post-release defects.

Hn2. There is no correlation between the executability of test classes and software quality, as measured by post-release defects.

An2. There is a correlation between the executability of test classes and software quality, as measured by post-release defects.

Regarding $\mathbf{RQ}_2$, we defined the following hypotheses:

Hn3. There is no correlation between test code metrics and software quality, as measured by post-release defects.

An3. There is a correlation between test code metrics and software quality, as measured by post-release defects.

Hn4. There is no correlation between test smells and software quality, as measured by post-release defects.

An4. There is a correlation between test smells and software quality, as measured by post-release defects.

Finally, we report below the hypotheses to address $\mathbf{RQ}_3$:

Hn5. There is no correlation between code coverage metrics and software quality, as measured by post-release defects.

An5. There is a correlation between code coverage metrics and software quality, as measured by post-release defects.

Hn6. There is no correlation between mutation coverage and software quality, as measured by post-release defects.

An6. There is a correlation between mutation coverage and software quality, as measured by post-release defects.

To test the hypotheses, we built a statistical model relating the independent and confounding factors to the post-release defects. A first key design decision regarded the proper choice of the statistical approach to fit our observations. We built a Generalized Linear Model (GLM) [63]. This method models the relationship between a scalar response (i.e., number of post-release defects in our case) and one or more explanatory variables (i.e., the selected set of independent and confounding factors) by fitting a linear function whose unknown model parameters are estimated from the data. We use the *'Gaussian'* family when implementing the model.

We relied on this approach for two main reasons. First, it simultaneously analyzes the effects of both confounding and independent variables on the response variable [35]. Second, it does not require distribution of data to be normal: indeed, in our case, the Shapiro-Wilk test [86] rejects the null-hypothesis, i.e., our data is not normally distributed.

To avoid multicollinearity [65], which may bias the interpretation of the results [58, 65], we first applied a hierarchical clustering based on the Spearman's rank correlation coefficient [91] of the studied variables (done using the `varclus` function available in the `R` statistical toolkit[18]), then, if two variables had a correlation higher than 0.6, we excluded the more complex one from the model. We interpreted the output of the Generalized Linear Model by considering the statistical significant codes it assigns to each explanatory variable, i.e., if a certain variable is deemed as statistically significant, the chances of the effect on the number of post-release defects being random is sufficiently low. We also computed the Adjusted R-squared [23] to assess the goodness of fit of the model, a metric indicating how close the data is to the fitted regression line.

[18] https://bit.ly/2YFltBU

4.2 Analysis of the Results

For each **RQ**, we build the models in a progressive manner, so that we can measure the explanatory power of different factors step-by-step. In particular, for each analysis we define three models: (i) the first only considers the effects of test-related factors, (ii) the second considering both production and test metrics, and (iii) a full model that includes production metrics, test-related factors and the selected process metric.

4.2.1 RQ_1. The presence and executability of tests

Table 4 reports the results of our first analysis. From the total set of variables employed within the model, we had to exclude (i) PWMC and PEC because they were too correlated with PLOC and (ii) the one signaling the presence of the *Spaghetti Code* smell, which in this case had high correlation with the presence of *Blob* instances. In conclusion, the model was composed of a total of seven metrics whose distributions are described in Table 5. The Adjusted R-squared measured 0.264: the value can be considered "moderate" [15, 17], namely the statistical model can fit the data with a moderate precision: $\approx 85\%$ of variation is still unexplained. Results of the first model led us to reject the two null hypotheses related to our first research question (i.e., Hn1 and Hn2) in favor of the alternative hypotheses (i.e., An1 and An2), indicating that the presence and the executability of test classes have a correlation with the number of post-release defects. However, when adding confounding factors, the hypotheses cannot be rejected nor confirmed.

Table 4 Results for $\mathbf{RQ}_1$ - The impact of the presence and executability of tests on the number of post-release defects. $N = 1,457$

	Test			Test + Prod.			Full		
	Estimate	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**
Intercept	0.11	0.05	*	-0.03	0.05		-0.17	0.04	***
is-tested	0.57	0.12	***	0.14	0.12		-0.09	0.11	
are-tests-executable	-0.49	0.13	***	-0.23	0.12	.	-0.05	0.11	
PLOC				0.00	0.00	***	0.00	0.00	
isGodClass				0.24	0.16		0.17	0.15	
isClassDataShouldBePrivate				0.39	0.38		0.70	0.34	
isComplexClass				-1.12	0.30	***	-0.29	0.27	
pre-release changes							0.05	0.00	***
pre-release defects							0.10	0.01	***
Multiple R-squared: 0.268; Adjusted R-squared: 0.264									
significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1									

Looking at the table, the variable which mostly influences post-release defects is the number of *pre-release changes*. Thus, we can confirm previous findings in the field [34, 44] on the relevance of this variable: the information coming from the past history of a class is a valuable predictor of its future quality. At the same time, the contribution given by this metric somehow hides the value of other product-based confounding variables: more specifically, factors like production code size and code smells—that were found to be highly

Table 5 Descriptive statistics for variables used in $\mathbf{RQ}_1$. $N = 1,457$

Variable name	Minimum	Maximum	Mean	SD
PLOC	2.00	6291.00	211.00	359.90
isGodClass	0.00	1.00	0.10	0.30
isClassDataShouldBePrivate	0.00	1.00	0.01	0.09
isComplexClass	0.00	1.00	0.02	0.14
is-tested	0.00	1.00	0.49	0.50
are-tests-executable	0.00	1.00	0.38	0.49
pre-release changes	0.00	201.00	7.32	12.26
pre-release defects	0.00	35.00	1.03	3.37
post-release defects	0.00	23.00	0.20	1.33

relevant to explain the future defect-proneness of source code [43, 59, 70]—are subsumed by this change history-based metric. Indeed, observing the results of the other two models which do not contain the process metric, we notice that some aspects related to the production code result to be highly significant (i.e., PLOC, Complex Class).

In the full model, the two independent variables selected for $\mathbf{RQ}_1$, i.e., *is-tested* and *are-tests-executable* are not statistically related to the number of post-release defects that will incur in source code classes. This result suggests that the mere existence of tests and/or their executability does not affect the number of post-release defects in the exercised production code.

From another perspective, our results can be also interpreted as a sign that the *quantity* of tests is not enough, and that perhaps their *quality* can serve as better indicators of post-release defects. In the next research question, we investigate whether the *quality* of tests is related to post-release defects.

RQ1 - Findings

Neither the executability nor the presence of tests are statistically significant variables to explain post-release defects when other confounding factors are taken under consideration. We confirm that the number of pre-release changes has the highest explanatory power.

4.2.2 $\mathbf{RQ}_2$. The impact of static test code indicators

While in the first research question we considered the entire set of instances belonging to our dataset, in $\mathbf{RQ}_2$ we have to consider on a smaller set of cases (as also seen in Figure 4), because we focus on the relation of statically computable test-related indicators to post-release defects, therefore the presence of tests is required to compute these indicators. The dataset we consider for the second research question has 774 observations and 184 post-release defects (as opposed to the 231 of the dataset used in $\mathbf{RQ}_1$). The descriptive statistics for all the variables are reported in Table 6. When removing untested classes from the dataset, the mean of the considered process metric (i.e., *'pre-release changes'*)

increases (see Tables 5 and 6). This may suggest that the number of changes tends to be lower when tests are not available, perhaps because developers are less confident with modifying the source code in such a circumstance or because tests are added when more changes are needed on certain files.

The multicollinearity analysis led us to remove (i) PWMC, PEC, TWMC, and TEC, as they were too correlated with the PLOC and TLOC metrics, and (ii) the variable related to the *Spaghetti Code* smell, as this still has a high correlation with the *Blob* code smell. Therefore, the model was composed of a total of 12 variables and reached an Adjusted-R squared of 0.282 (moderate).[19] This means that the model only explains $\approx 15\%$ of variation.

In the first place, the results, reported in Table 7, show that TLOC, is highly significant when test-related factors are considered in isolation, thus allowing to reject Hn3 in favor of An3. However, this relation cannot be extended to the full model since the addition of confounding factors led to a decrease of the significance of test code metrics, namely Hn3 cannot be rejected nor confirmed.

Table 6 Descriptive statistics for variables used in $\mathbf{RQ}_2$. $N = 774$

Variable name	Minimum	Maximum	Mean	SD
PLOC	13.00	6291.00	311.00	460.00
isGodClass	0.00	1.00	0.17	0.38
isClassDataShouldBePrivate	0.00	1.00	0.01	0.12
isComplexClass	0.00	1.00	0.04	0.19
TLOC	5.00	2210.00	168.00	248.10
Assertion Density	0.00	0.83	0.19	0.15
isAssertionRoulette	0.00	1.00	0.87	0.34
isEagerTest	0.00	1.00	0.61	0.49
isMysteryGuest	0.00	1.00	0.07	0.26
isResourceOptimism	0.00	1.00	0.02	0.14
isIndirectTesting	0.00	1.00	0.06	0.24
pre-release changes	1.00	201.00	10.00	16.21
pre-release defects	0.00	35.00	1.73	4.54
post-release defects	0.00	23.00	0.29	1.65

The results of the full model confirm the previous ones: there is a strong relation between the process metric we considered (i.e., pre-release changes) and post-release defects even when adding factors quantifying the properties of test code. On the one hand, this finding reinforces the idea that the change process underwent by production classes is among the most valid indicators for software quality. On the other hand, statically computable properties of test code do not impact the future defect-proneness of production classes.

The only exception is the variable measuring the presence of the test smell *Mystery Guest*, which allowed to reject the null hypothesis Hn4 in favor of the alternative hypothesis An4. *Mystery Guest* is a test smell that appears when

[19] Note that the three statistical models for the different research questions are not comparable in terms of R^2, since they operate on different datasets (see Figure 4). We report the values for the R^2 only to give an idea of the statistical models' explanatory power.

Table 7 Results for $\mathbf{RQ}_2$ - The impact of static test-related factors on the number of post-release defects. $N = 774$

	Test			Test + Prod.			Full		
	Estimate	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**
Intercept	-0.02	0.18		-0.08	0.18		-0.19	0.16	
TLOC	0.00	0.00	***	0.00	0.00		-0.00	0.00	
Assertion Density	0.16	0.44		0.02	0.43		-0.38	0.39	
isAssertionRoulette	-0.06	0.20		-0.04	0.19		0.05	0.17	
isEagerTest	0.12	0.14		0.03	0.14		-0.05	0.13	
isMysteryGuest	-0.19	0.28		-0.20	0.28		-0.54	0.25	*
isResourceOptimism	0.73	0.53		0.54	0.52		-0.22	0.47	
isIndirectTesting	-0.01	0.27		-0.04	0.27		0.09	0.24	
PLOC				0.00	0.00	***	0.00	0.00	
isGodClass				0.36	0.23		0.32	0.21	
isClassDataShouldBePrivate				0.27	0.56		0.93	0.50	.
isComplexClass				-0.97	0.41	*	-0.22	0.38	
pre-release changes							0.03	0.00	***
pre-release defects							0.09	0.02	***
Multiple R-squared: 0.294; Adjusted R-squared: 0.282									

significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1

a test relies on external resources (e.g., files) [97]. To understand the reasons behind this finding, the first two authors of this paper jointly looked at the tests affected by this smell, trying to understand the characteristics of those tests that could justify a similar result. In the end, the two researchers come to the conclusion that there may exist an indirect relation between this smell and production code quality. Specifically, one of the main negative consequences of having *Mystery Guest* instances is the non-deterministic behavior of the affected test code [97]. Intuitively, test classes that intermittently pass/fail cannot properly exercise the corresponding production code and find defects: thus, one likely reason behind the achieved result is the direct relation between this test smell and test flakiness [97], which therefore turns to be indirect when considering *Mystery Guest* and post-release defects. To some extent, our findings also confirm what reported by Spadini et al. [90] on the relation between test smells and defect-proneness of production code. This observation has to be confirmed through further empirical investigations.

The results obtained when considering the first two models (i.e., *'test'*, *'test + prod'*) also confirm the ones reported in $\mathbf{RQ}_1$; indeed, PLOC and presence of Complex Class instances are statistically significant factors only when the process variable is not taken into account.

RQ2 - Findings

The size of the test classes relates to post-release defects only if no production and process metrics are considered. We also found that *Mystery Guest* is a statistically significant factor, but a fine-grained analysis only highlighted its possible indirect relation to software quality.

4.2.3 $\boldsymbol{RQ_3}$*. The impact of dynamic test code indicators*

In the last research question, we measure how dynamically computable test code indicators are related to post-release defects. To compute these indicators,

Table 8 Descriptive statistics for variables used in $\mathbf{RQ}_3$. $N = 577$

Variable name	Minimum	Maximum	Mean	SD
PLOC	13.00	5077.00	259.00	342.20
isGodClass	0.00	1.00	0.12	0.38
isClassDataShouldBePrivate	0.00	1.00	0.01	0.11
isComplexClass	0.00	1.00	0.02	0.14
TLOC	5.00	2210.00	135.90	194.40
Assertion Density	0.00	0.83	0.20	0.16
isAssertionRoulette	0.00	1.00	0.86	0.34
isEagerTest	0.00	1.00	0.62	0.49
isMysteryGuest	0.00	1.00	0.06	0.23
isResourceOptimism	0.00	1.00	0.02	0.12
isIndirectTesting	0.00	1.00	0.04	0.20
Line Coverage	0.00	1.00	0.90	0.14
Branch Coverage	0.00	1.00	0.75	0.32
Mutation Coverage	0.00	1.00	0.70	0.32
pre-release changes	1.00	139.00	8.49	10.91
pre-release defects	0.00	29.00	1.37	3.66
post-release defects	0.00	23.00	0.19	1.23

Table 9 Results for $\mathbf{RQ}_3$ - The impact of dynamic test-related factors on the number of post-release defects. $N = 577$

	Test			Test + Prod.			Full		
	Estimate	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**	**Estimate**	**S.E.**	**Sig.**
Intercept	0.17	0.38		-0.16	0.37		-0.18	0.36	
Line Coverage	0.27	0.45		0.41	0.44		0.21	0.43	
Branch Coverage	-0.08	0.17		-0.08	0.17		-0.17	0.16	
Mutation Coverage	-0.55	0.18	**	-0.35	0.19	.	-0.12	0.18	
LOC (test suite)	0.00	0.00	***	-0.00	0.00		-0.00	0.00	*
Assertion Density	0.38	0.36		0.15	0.35		-0.12	0.34	
isAssertionRoulette	-0.10	0.17		-0.04	0.16		0.01	0.16	
isEagerTest	0.09	0.12		0.03	0.12		-0.04	0.11	
isMysteryGuest	-0.14	0.27		-0.02	0.27		-0.22	0.26	
isResourceOptimism	0.67	0.50		0.54	0.49		0.32	0.47	
isIndirectTesting	0.23	0.26		0.16	0.26		0.18	0.25	
LOC (production class)				0.00	0.00	***	0.00	0.00	**
isGodClass				-0.08	0.23		-0.14	0.22	
isClassDataShouldBePrivate				1.22	0.54	*	1.49	0.52	**
isComplexClass				0.26	0.45		0.37	0.43	
pre-release changes							0.03	0.01	***
pre-release defects							0.04	0.02	*
							Multiple R-squared: 0.244; Adjusted R-squared: 0.225		
							significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1		

we needed to analyze tests that are executable. This restricted the scope to a dataset including 103 post-release defects. To study the effect of dynamic test code indicators, we computed and integrated them in the model coming from $\mathbf{RQ}_2$, thus building a Generalized Linear Model with 15 variables whose descriptive statistics are reported in Table 8. To avoid collinearity, we had to exclude PWMC, PEC, TWMC, TEC, and *Spaghetti Code*. The goodness of fit of the resulting full model was 0.225, which indicates that the model has a weak explanatory power—this because the percent by which the standard deviation of the errors is less than the standard deviation of the dependent variable is pretty low: $\approx 13\%$.

Table 9 reports the results. When analyzing the model that only includes test-related factors, *Test LOCs* and *Mutation Coverage* are statistically significant. On the one hand, this result may indicate that larger test classes, which likely contain more tests, represent a better guard to the introduction of post-release defects. On the other hand, mutation coverage measures the ability of tests to identify artificially created defects; looking at the sign of the relation, our findings suggest that having a lower mutation coverage is related to a higher number of post-release defects, which is expected given the goal of mutation testing. Nevertheless, this variable is significant only when considering test-related factors in isolation, while its explanatory power decreases as additional factors are added to the model. In particular, the number of pre-release changes is among the most important factors related to software quality, while line and branch coverage do not relate to post-release defects, meaning that the amount of production code lines touched by a test does not reduce the likelihood to have faults in source code: this is in line with the recent findings of Kochhar et al. [48].

Line and branch coverage are not significant, regardless of the model considered. This does not allow us to reject the null hypothesis Hn5. Mutation coverage, instead, is significant only when test-related factors are considered in isolation. Also in this case we cannot reject the null hypothesis Hn6.

In this part of the dataset, with the addition of dynamic factors and process metrics, some of the previously not significant statically computable variables assumed a higher relevance. This is the case of LOC of production and test code. We extensively analyzed and discussed our data to understand the reasons behind this result and further discuss them in the following.

In $\mathbf{RQ}_2$, we considered both tests that could and could not be executed (see Figure 4): as such, the dataset possibly included non-executable tests having a large size which, clearly, could not exercise the production code and find defects. This might have limited the effect of TLOC on the dependent variable; conversely, when considering only executable test suites ($\mathbf{RQ}_3$), the variable turned to be significant. This statement is supported by the fact that 71% of the tests excluded from $\mathbf{RQ}_2$ have a LOC higher than the third quartile of the distribution of all test LOCs of the dataset.

A similar discussion can be done when considering the statistical significance of production code LOC. The dataset employed in $\mathbf{RQ}_3$ may have filtered out small classes exercised by non-executable tests that lead to post-release defects. To verify this hypothesis, we performed an additional analysis in which we assessed how the results of $\mathbf{RQ}_3$ change when running a model only based on statically computable test code indicators (i.e., the setting used in $\mathbf{RQ}_2$): as a result, we found that PLOC remains significant, meaning that the different statistical findings are indeed due to the specific composition of the exploited dataset. Another interesting observation concerns the relation between test and production size metrics. The directions of the two distributions (i.e., column "Estimate" in Table 9) are opposite, which means that: (i) the larger the TLOCs, the fewer the number of post-release defects, and (ii) the larger the PLOCs, the higher the number of post-release defects. This result is quite expected and can

be better explained analyzing two relevant qualitative examples. The first one is the class `ClassDerivativeStructure` belonging to the Commons-Math project; it shows a high number of PLOCs (i.e., 1,011) but at the same time a high number of TLOCs (i.e., 1,172). This class has no post-release defects, perhaps due to the robustness of the test suite. The second example is the class `BaseGenericObjectPool` in the project Commons-Pool; like the previous one, it is characterized by a high PLOCs (i.e., 849) but, the low number of TLOCs (i.e., 43) may have led to 23 post-release defects (the maximum number of defects among the instances we analyzed). These two examples, together with the results of the statistical model, suggest that the size of test suites can be a proxy metric to assess how robust a test is. Differently from the other research questions, the considered variables remain significant across the four models. Indeed, also adding the process metrics, the PLOC and *'isClassDataShouldBePrivate'* are still significant.

RQ3 - Findings

Mutation coverage statistically relates to post-release defects only when test-related factors are considered in isolation. When considering both static and dynamic test code indicators, we observed production and test code size to be statistically significant in explaining post-release defects. Furthermore, our findings suggest that TLOC can be a proxy metric to assess the quality of the test.

5 Discussion, Implications, and Threats to Validity

In this section we further discuss the main outcomes of our work, describe the implications of the study for both researchers and practitioners, and examine the threats that might have influenced the validity of the results.

5.1 Discussion

The results of the study provided two main findings to be further discussed.

On the (limited) importance of test-related factors for software code quality. The main outcome of our research reports that—surprisingly—most of the considered test-related factors do not have a significant explanatory power with respect to post-release defects. Despite this could seem strange, it is possible to reason on why this could happen. Let consider a scenario in which tests have good quality and effectiveness: these tests are likely to accurately identify defects present in the same snapshot of the production code, however they do not necessarily predict the future defect-proneness of the class under test. So, probably the test-related factors commonly known and used in literature are not appropriate enough to catch the defect proneness of the

exercised code. Proposing new metrics able to describe the relation between tests and software quality could be an interesting cue for future works.

From another point of view, there are some exceptions that may allow us to claim that keeping test code design under control might still help developers in reducing the number of post-release defects. In the first place, looking at the results of $\mathbf{RQ}_3$, the test LOC metric not only appears as highly significant, but it is also inversely proportional to the dependent variable: this indicates that larger tests (that are likely to exercise deeper the production code) reduce the risk of having defects in future versions of the system. Thus, our findings seem to indicate that the lines of code of a test suite may represent a proxy measure for test-code effectiveness.

Furthermore, while other test-related factors investigated in the study (e.g., line coverage or assertion density) are not correlated enough to test LOC to cause collinearity, it is reasonable to believe that they have some sort of relations with the size of the test: for instance, the assertion density generally tends to increase with the size of the test [50]. On the one hand, these relations should be further assessed in the future. On the other hand, this observation may further suggest that high-quality tests lead to post-release defect reduction: the results achieved in $\mathbf{RQ}_2$ on the role of test smells, particularly *Mystery Guest*, also go toward this conclusion.

In order to better comment on our results, we performed an additional qualitative analysis in which we contacted the top contributors of the considered projects. In so doing, we followed a similar experimental design as Mäntylä et al. [54]. In particular, we sent direct e-mails to the two top developers of the systems, i.e., the two having the highest amount of commits, asking them to comment on our findings and provide feedback on some boundary cases we discovered when analyzing their systems—this means that developers were inquired only on the matters related to their own projects. Unfortunately, we received an answer for just one of the considered systems—even tough they were insightful to better contextualize and understand the findings of the study. The answers was related to Commons-Pool. In this specific case, we asked the developer to comment on the release 2.3 of the project, in which the class named `BaseGenericObjectPool` had 849 lines of code, while the corresponding test suite `BaseGenericObjectPoolTest` had just 43 lines of code. After release 2.3, the class `BaseGenericObjectPool` had 23 defects. At a first sight, this may suggest that the test suite was not robust enough in preventing or diagnosing the introduction of defects. However, the developer found that just considering the test suite `BaseGenericObjectPoolTest` could potentially be not enough. He pointed out to us that when code is refactored, the tests are left in the original test suites to help detect regressions during the refactoring. So, there could exist a subset of tests in other classes, that we did not consider, which exercise the production class `BaseGenericObjectPool`—the test-to-code traceability technique exploited in the study may have under-estimated the number of tests connected to the production class.

To sum up, our findings reveal that the problem of understanding the effect of tests on post-release defects is still open and would require further

investigations—especially in the lights of the potential threat to validity related to the test-to-code traceability technique raised by the involved developer (Section 5.3 further discusses this point). At the same time, the results provide some hints of the importance of (i) test-related factors for software quality, even though other aspects, e.g., the number of changes to production code, are still primarily connected to post-release defects and (ii) continuously keep test suites up to date with the changes applied to production code.

On the comparison with previous studies. As explained in Section 2, a number of researchers have investigated the role of test-related factors on post-release defects in the past, finding them as highly relevant. While our results do not tell the opposite, we found that most of the considered factors have a lower explanatory power than the one previously reported.

The key to explain the difference between our outcomes and the ones previously provided is in the presence of confounding factors that possibly balanced the effect of test-related factors. This is particularly true in the case of LOCs of production class and pre-release changes, that are the most relevant metrics to explain the future defect-proneness of source code and are directly proportional to the dependent variable, i.e., the higher the size and the number of pre-release changes, the higher the number of post-release defects. This suggests a pretty straightforward interpretation: classes having large size or being involved in several changes over the history are more prone to have defects in the future.

5.2 Implications

Our results have a number of implications for both research community and practitioners.

Keep the change process under control. The most important finding of our study is the very high influence of pre-release changes on post-release defects. This relation indicates that the change frequency of classes impacts the future defect-proneness of production code more than other aspects, confirming previous findings on the relationship between change- and defect-proneness [16, 37]. In this respect, it may be possible that high-quality pre-release changes prevent the emergence of post-release defects: in the context of our $\mathbf{RQ}_3$, we indeed noticed that the LOC of production code—which has been often used as a proxy metric for code quality—and pre-release defects are statistically significant factors for the future defect-proneness of source code. Based on these observations, we can claim that keeping the change process under control would be worthwhile and that the definition of mechanisms supporting developers when dealing with software evolution represents a key challenge for the research community. While a number of attempts in this direction have been performed during the last years, e.g., through the definition of of just-in-time quality assurance mechanisms [11, 41, 75, 73], we believe that further research effort should be invested. In particular, most

of the approaches developed so far should be considered as prototypes and, as such, are still not mature enough to be used in practice. For example, researchers have been working on just-in-time defect prediction models (e.g., [41, 75]), but up to now there are no fully-available tools that enable their practical usage. This clearly represents a key threat to the adoption of these tools in practice that the research community should investigate more. As a consequence, we argue that continuous integration pipelines as well as typical software development practices should be empowered with additional instruments that allow developers to promptly assess the quality of the changes made on production code: for instance, we refer to defect localization tools that can be integrated within CI environments or code smell detectors and refactoring recommenders that allow an agile quality improvement of source code during the code review process.
At the same time, tool vendors have been spending effort in providing developers with tools that can help them spotting defects. The main outcome is represented by automated static analysis tools, which are generally used in open-source systems as highlighted by a number of papers in literature [101, 100, 103]. One of these tools is FindBugs, which is the one employed by the Apache Software Foundation and, as a consequence, by the projects considered in our paper (this tool is configured within the Apache Continuum server they have in place). On the one hand, static analysis tools suffer from a high rate of false positive alerts [40]: this aspect has the effect of reducing the trust of developers with respect to the outcome of these tools, possibly leading them to ignore relevant defect warnings. On the other hand, it has been shown that open-source systems (including those of the Apache Commons family) do not apply continuous code quality practices [99], meaning that they do not run quality checks at every build they do: this is an additional limitation of the current quality assurance practices. According to these observations, our work further stimulates the research effort around the definition of techniques able to reduce the number of false positives given by static analysis tools as well as mechanisms enabling the adoption of continuous code quality.

Test-related factors and defect prediction. The results of the study revealed that, in some cases, test-related factors are related to post-release defects. While we cannot speculate on whether there exist specific types of defects that can be better analyzed through the exploitation of test-related factors, we still see some value in this finding. More specifically, from our study we observed that test-related factors become relevant especially when process metrics are not considered. This represents an interesting case for defect prediction: indeed, new projects interested in deploying these models might not have enough historical data to enable the computation of process metrics. In these cases, there are two solutions. On the one hand, developers may rely on cross-project information to train defect prediction models [106]: nevertheless, the adoption of this strategy does not still provide accurate results, hence limiting its applicability. On the other hand, developers can

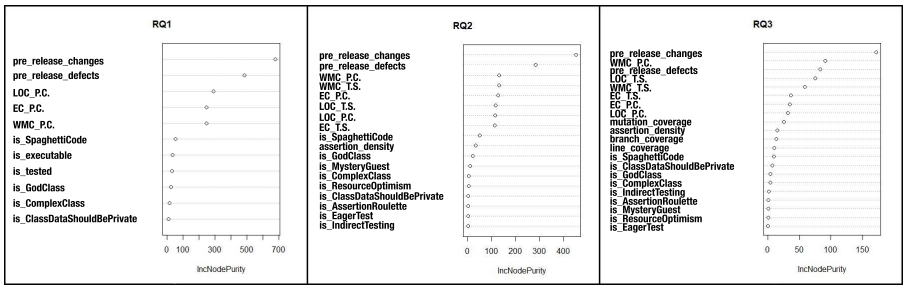

Figure 5 The results for Random Forest.

create prediction models based on product information coming from the analysis of their own systems: our results can be useful in this context, as test-related factors may complement other product metrics and potentially improve the quality of the predictions. In the recent past, researchers have started looking at the role of tests in defect prediction [5], however we believe that our study may inspire further research on the matter, especially based on the factors that turned to be important for post-release defects, e.g., test size or presence of five test smells considered in the paper.

Test-related factors and automatic test case generation. Our findings point out that other test-related factors, namely test size and test smells, are more related to post-release defects than metrics generally considered relevant, i.e., code or mutation coverage. This seems to suggest that the design of test suites matter. This may possibly pave the way for the next generation of automatic test case techniques that do not consider anymore (or decrease the importance of) code and mutation coverage as main metrics to optimize during the creation process: such a generation mechanism would possibly allow automatic tools to focus on the creation of tests around factors that are more connected to post-release defects (as also proved by Kochhar et al. [48]). Furthermore, our results also highlight the existence of production-related factors, and specifically production code size and presence of code smells, that have a relation with post-release defects. It would be worth to consider how these production-related factors can contribute to the generation of effective test classes: for example, existing automated tools may exploit these metrics within their fitness function and balance them with other metrics with the aim of refining or further optimizing the generated test suites around the metrics that are more connected to post-release defects.

5.3 Threats to Validity

The results of our study might have been biased by a number of factors. In this section, we overview the main threats to validity and how we mitigated them.

Construct validity. Threats in this category are concerned with the relation between theory and observation. A key point in this regard is related to the accuracy of the tools used to compute the explanatory and dependent variables of the study. First, we are aware of the possible limitations of the SZZ algorithm highlighted in recent works [84]: in this respect, we conducted a manual validation of the results of the SZZ algorithm that aimed at excluding false positives. Of course, we are aware that this analysis could not cope with false negatives: we made all data of our study available to make other researchers able to replicate and possibly extend our work with additional findings. Similarly, possible issues with the measurement of the exploited independent variables have been mitigated by the selection of tools that are (i) well established in the field (e.g., DECOR [56] for the detection of code smells) and (ii) accurate enough for conducting our study, according to the manual validations conducted in the context of our work. Also in this case, however, it is worth mentioning that the manual analyses could only deal with false positives but not with false negatives.

A partially different discussion should be made when considering test smells. To detect them, we relied on the detector made available by Bavota et al. [4]. While previous studies have shown that its F-Measure is close to 86% [4, 66, 72], the detector could have had a different accuracy in our context. Recognizing this as a possible threat to validity, we conducted an additional investigation aimed at measuring the precision of the detector.[20] Unlike the case of code smells, we could not validate all the 1,217 instances output by the test smell detector as a manual analysis would have been excessively expensive. Instead, we focused on a stratified statistically significant sample (confidence level=95%, confidence interval=5%) composed of 169 instances. The task was jointly conducted by the two first authors of the paper and consisted of assessing whether each test smell candidate presented a certain design issue. At the end of the process, 86% of the instances were considered as real test smells - thus confirming the high precision of the detector.

Another discussion point relates to the methodology employed to link production classes to test cases: in particular, we employed a traceability technique based on naming conventions, i.e., it identifies the test corresponding to a certain production class by looking at the name of the test and verifying whether it is the same as the production class expect with the prefix '`Test`'. While the accuracy of the technique has been previously assessed [98] showing a good compromise between accuracy and scalability, the linking procedure may have introduced some bias in cases tests exercising a production class are not all included in the test suite retrieved by the technique but put in other test suites. In our case, this may have been happened, as mentioned by the interviewed developer of APACHE COMMONS-POOL who commented on our findings in the context of our additional qualitative analysis (see Section 5.1).

There are two observations to make with respect to this potential bias. First, it has been pointed out by only one developer and was related to only one of

[20] The recall cannot be assessed because of the lack of an oracle.

the considered projects: as such, we are not able to estimate the extent of this bias in other systems as well as to verify whether this may have represented a general problem for Commons-Pool or if it was instead focused on a subset of classes of the project—it is worth noting that a manual examination of this bias would have not only been prohibitively expensive, but also error-prone given our lack of expertise on the project. Second, we could not identify an alternative traceability technique which may have provided better results than the one employed: indeed, while some more sophisticated test-to-code traceability techniques have been proposed [74], these are likely to suffer from similar issues as the one based on naming convention. As an example, the slicing-based approach proposed by Qusef et al. [80] exploits slicing and conceptual coupling to identify the set of test suites associated with a production class. By design, this approach may have higher recall, since it is able to model the case in which more test suites exist for a production class. At the same time, however, this may not be enough. The involved developer mentioned a finer-grained problem where specific test cases are included in other suites, as opposed to the existence of multiple test suites for a production class. As such, the technique by Qusef et al. [80] may lead to overestimate the number of tests for a certain class, decreasing the precision of the analysis. In other words, such a fine-grained linking between test suites and production classes would have needed a traceability approach able to cluster the test cases connected to a production class: unfortunately, to the best of our knowledge, such a technique is not available in literature—we hope that this additional finding may serve as an input for the software traceability research community.

Finally, to extract a comprehensive list of test-related factors to experiment, we applied a MLR [31]. In this regard, possible threats refer to the soundness and completeness of the review. With respect to the former, the first two authors of this paper followed well-established guidelines [46, 102] to search, analyze, and select relevant sources; moreover, the joint work conducted by the two authors have reduced the risk of subjective evaluations of the resources to include as well as allowed a quick solving of possible disagreements. As for the latter, we defined a search query targeting the research goals of the paper; at the same time, we targeted databases that allow searching for most of the white papers published in our community. Furthermore, we analyzed all the relevant Google pages when gathering gray literature, also performing it using the incognito mode to avoid biases due to previous navigation history.

Internal validity. Threats to internal validity concern with intrinsic factors of our study that could have influenced the reported results. In this regard, there are some intrinsic issues when computing some of the test-related factors, like mutation coverage. In particular, there are two relevant aspects when measuring this metric: the problem of equivalent mutants and the one of live mutants. As for the former, it arises when two generated mutants are semantically equivalent. Unfortunately, determining whether a mutant is equivalent is an undecidable problem. Furthermore, detecting them is also hard - there is still no mature tool available for this task [53] - and computationally expensive

[64]. For these reasons, equivalent mutants represent a common threat to the validity of the results of studies concerned with mutation testing. In our study setup we took into account the equivalent mutants problem when selecting the mutation testing tool. Among the available ones, PiTest has shown better performance than others: specifically, less than 20% of the mutants generated by the tool are deemed to be equivalent, which represents an important step forward in the context of mutation testing [26]. Other control mechanisms, e.g., manual removal of equivalent mutants, would not be feasible in our case because of the high number of mutants generated by PiTest.

As for the live mutants, these represent the mutants generated by the tool but not detected by the available tests. Live mutants allow the mutation score computation (i.e., mutants detected over mutants generated). On average, these mutants represent around 30% of all mutants generated, as shown in Section 4.2.3, meaning that most of the tests in our dataset have a rather high mutation score. This suggests that the study takes into account valuable tests that can actually be used to investigate post-release defects.

Conclusion validity. As for the relationship between treatment and outcome, a first possible threat is connected to the statistical models built in our **RQ**s. Throughout our research, we controlled the impact of test-related factors for possible confounding effects due to the characteristics of production code, considering both product and process metrics that have been shown to be connected with the dependent variable. Furthermore, depending on the specific research question posed, we included in the statistical model the most suited test-related factors. Another threat is related to the actual suitability of the employed statistical method, i.e., Generalized Linear Model. In this regard, before selecting it we verified the assumptions that the model makes on the underlying data. Nevertheless, it may still be possible that the statistically significant variables discovered through the use of linear regression may be due to the specific data manipulation and analysis done by the statistical model [38]. To better investigate this aspect and test the robustness of our findings, we completely repeated our analyses by using a different statistical model, namely Random Forest [85], that makes no assumptions on the underlying data and is robust to overfitting [38]. A summary of the results of this additional analysis is shown in Figure 5, where we show the relevance of the variables experimented in each **RQ** with respect to post-release defects. Specifically, we computed the Mean Decrease in Impurity (also known as the Gini index) [85], an entropy-based metric that is used by Random Forest to elicit the variables providing the higher contributions to the explanation of the dependent variable. As shown, the results are in line with those reported in Section 4.2, with (i) pre-release changes and production class LOCs being the most important variables to explain post-release defects and (ii) the other explanatory variables having a significance similar to the one identified by Generalized Linear Model. This strengthen our confidence on the validity of the results.

External validity. With respect to the generalizability of our findings, we analyzed eight open-source Java projects. Analyzing only a small number of

systems could threat the external validity of our study. However, during the context selection, we had to deal with a set of constraints that have significantly limited the list of candidate systems. In particular, we restricted our search to Maven projects with no sub-modules and a standard Maven structure. We made this choice to allow the employment of the PiTest command-line tool for the calculation of the dynamic factors (i.e., line and mutation coverage). Nevertheless, further replications of our study, conducted in different contexts overcoming the constraints mentioned above (e.g., by aggregating the results of multiple submodules) might be worthwhile to corroborate our findings.

Moreover, it is worth remarking that the statistical models were built over the specific independent, control, and dependent variables adopted in the study. Despite our effort in taking into account all factors known to have an impact on post-release defects, we are aware that different results may arise when considering different/additional variables. Similarly, our findings on the relation between test smells and software quality are deemed to be valid for the five test smell types considered in the paper: other results may be achieved with other design issues.

Another aspect to consider when interpreting our results is that some post-release defects may be discovered with unit testing, while others can be found only at higher-levels (e.g., with integration or system testing) [21]. We are aware of this point and recognize that our study is limited to the analysis of the behavior and the relation that unit tests have with post-release defects. Replications of our study targeting different test levels would provide a more comprehensive view of how tests can forecast post-release defects. On a similar note, our study investigated the role of the tests actually available in the considered software projects: it may be possible that different results could be achieved in cases where the diversity of test cases, i.e., the extent to which a test is different from the others in the suite [24, 25], is higher. Understanding the impact of diversity on both test and production code quality is part of our future investigations.

6 Conclusion

In this paper, we have presented an empirical study we conducted to investigate the role of test-related factors on software quality, operationalized as the number of post-release defects of production source code files. We considered eight Apache-Commons systems as our case study, as they satisfy important selection criteria. We found that neither the executability nor the presence of tests is a significant factor to explain post-release defects in a statistical model. In addition, other, finer-grained test-related factors seem to have a limited effect on the number of post release defects, while we confirm the value of process factors as indicators of future software quality.

Our findings represent the input of our future research agenda, which is focused on enlarging the scope of our analyses and defining methods to better support developers when evolving test suites. Furthermore, we aim at

(i) replicating our study by considering a larger amount of systems as well as integration and system tests and (iii) considering the role of test diversity on software quality.

Acknowledgment

The authors would like to sincerely thank the Associate Editor and anonymous Reviewers for the insightful comments and feedback provided during the review process. Palomba gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Projects No. PZ00P2_186090 (TED).

References

1. Adamopoulos K, Harman M, Hierons RM (2004) How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: Genetic and evolutionary computation conference, Springer, pp 1338–1349
2. Andrews JH, Briand LC, Labiche Y, Namin AS (2006) Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Transactions on Software Engineering 32(8):608–624
3. Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Transactions on software engineering 22(10):751–761
4. Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2015) Are test smells really harmful? an empirical study. Empirical Software Engineering 20(4):1052–1094
5. Bowes D, Hall T, Harman M, Jia Y, Sarro F, Wu F (2016) Mutation-aware fault prediction. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, pp 330–341
6. Briand L, Bianculli D, Nejati S, Pastore F, Sabetzadeh M (2017) The case for context-driven software engineering research: Generalizability is overrated. IEEE Software 34(5):72–75
7. Brown WH, Malveau RC, McCormick HW, Mowbray TJ (1998) AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc.
8. Budgen D, Brereton P (2006) Performing systematic literature reviews in software engineering. In: Proceedings of the 28th international conference on Software engineering, pp 1051–1052
9. Cai X, Lyu MR (2007) Software reliability modeling with test coverage: Experimentation and measurement with a fault-tolerant software project. In: The 18th IEEE International Symposium on Software Reliability (ISSRE'07), IEEE, pp 17–26
10. Catolino G, Palomba F, De Lucia A, Ferrucci F, Zaidman A (2018) Enhancing change prediction models using developer-related factors. Journal of Systems and Software 143:14–28

11. Catolino G, Di Nucci D, Ferrucci F (2019) Cross-project just-in-time bug prediction for mobile apps: an empirical assessment. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, pp 99–110
12. Catolino G, Palomba F, Zaidman A, Ferrucci F (2019) How the experience of development teams relates to assertion density of test classes. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 223–234
13. Chen MH, Lyu MR, Wong WE (2001) Effect of code coverage on software reliability measurement. IEEE Transactions on reliability 50(2):165–170
14. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Transactions on software engineering 20(6):476–493
15. Chin WW, et al. (1998) The partial least squares approach to structural equation modeling. Modern methods for business research 295(2):295–336
16. Choudhary GR, Kumar S, Kumar K, Mishra A, Catal C (2018) Empirical analysis of change metrics for software fault prediction. Computers & Electrical Engineering 67:15–24
17. Cohen J (1992) Statistical power analysis, vol 1. Sage Publications Sage CA: Los Angeles, CA
18. da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2017) A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering 43(7):641–657
19. D'Ambros M, Lanza M, Robbes R (2009) On the relationship between change coupling and software defects. In: 2009 16th Working Conference on Reverse Engineering, IEEE, pp 135–144
20. D'Ambros M, Bacchelli A, Lanza M (2010) On the impact of design flaws on software defects. In: 2010 10th International Conference on Quality Software, IEEE, pp 23–31
21. Damm LO, Lundberg L, Wohlin C (2006) Faults-slip-through—a concept for measuring the efficiency of the test process. Software Process: Improvement and Practice 11(1):47–59
22. Di Nucci D, Palomba F, De Rosa G, Bavota G, Oliveto R, De Lucia A (2018) A developer centered bug prediction model. IEEE Transactions on Software Engineering 44(1):5–24
23. Draper NR, Smith H (2014) Applied regression analysis, vol 326. John Wiley & Sons
24. Feldt R (2014) Do system test cases grow old? In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, IEEE, pp 343–352
25. Feldt R, Torkar R, Gorschek T, Afzal W (2008) Searching for cognitively diverse tests: Towards universal test diversity metrics. In: 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, IEEE, pp 178–186
26. Fernandes L, Ribeiro M, Carvalho L, Gheyi R, Mongiovi M, Santos A, Cavalcanti A, Ferrari F, Maldonado JC (2017) Avoiding useless mutants.

In: Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp 187–198
27. Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, IEEE, pp 23–32
28. Fowler M (2018) Refactoring: improving the design of existing code. Addison-Wesley Professional
29. Fregnan E, Baum T, Palomba F, Bacchelli A (2019) A survey on software coupling relations and tools. Information and Software Technology 107:159–178
30. Garousi V, Küçük B (2018) Smells in software test code: A survey of knowledge in industry and academia. Journal of systems and software 138:52–81
31. Garousi V, Felderer M, Mäntylä MV (2016) The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In: Proceedings of the 20th international conference on evaluation and assessment in software engineering, pp 1–6
32. Grano G, Palomba F, Gall HC (2019) Lightweight assessment of test-case effectiveness using source-code-quality indicators. IEEE Transactions on Software Engineering p in press
33. Grano G, De Iaco C, Palomba F, Gall HC (2020) Pizza versus pinsa: On the perception and measurability of unit test code quality p to appear
34. Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Transactions on software engineering 26(7):653–661
35. Halekoh U, Højsgaard S, Yan J, et al. (2006) The r package geepack for generalized estimating equations. Journal of Statistical Software 15(2):1–11
36. Hall T, Zhang M, Bowes D, Sun Y (2014) Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology (TOSEM) 23(4):33
37. Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, pp 78–88
38. Hayes AF (2009) Beyond baron and kenny: Statistical mediation analysis in the new millennium. Communication monographs 76(4):408–420
39. Islam C, Babar MA, Nepal S (2019) A multi-vocal review of security orchestration. ACM Computing Surveys (CSUR) 52(2):1–45
40. Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 672–681
41. Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 39(6):757–773, DOI 10.1109/

TSE.2012.70
42. Keele S, et al. (2007) Guidelines for performing systematic literature reviews in software engineering. Tech. rep., Technical report, Ver. 2.3 EBSE Technical Report. EBSE
43. Khomh F, Di Penta M, Guéhéneuc YG, Antoniol G (2012) An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering 17(3):243–275
44. Kim S, Zimmermann T, Whitehead Jr EJ, Zeller A (2007) Predicting faults from cached history. In: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, pp 489–498
45. Kim S, Whitehead EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering 34(2):181–196, DOI 10.1109/TSE.2007.70773
46. Kitchenham B (2004) Procedures for performing systematic reviews. Keele, UK, Keele University 33(2004):1–26
47. Knab P, Pinzger M, Bernstein A (2006) Predicting defect densities in source code files with decision tree learners. In: Proceedings of the 2006 international workshop on Mining software repositories, ACM, pp 119–125
48. Kochhar PS, Lo D, Lawall J, Nagappan N (2017) Code coverage and postrelease defects: A large-scale study on open source projects. IEEE Transactions on Reliability 66(4):1213–1228
49. Koru AG, Zhang D, El Emam K, Liu H (2009) An investigation into the functional form of the size-defect relationship for software modules. IEEE Transactions on Software Engineering 35(2):293–304
50. Kudrjavets G, Nagappan N, Ball T (2006) Assessing the relationship between software assertions and faults: An empirical investigation. In: 2006 17th International Symposium on Software Reliability Engineering, IEEE, pp 204–212
51. Li N, Praphamontripong U, Offutt J (2009) An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops, IEEE, pp 220–229
52. Lubsen Z, Zaidman A, Pinzger M (2009) Using association rules to study the co-evolution of production & test code. In: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on, IEEE, pp 151–154
53. Madeyski L, Orzeszyna W, Torkar R, Jozala M (2013) Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. IEEE Transactions on Software Engineering 40(1):23–42
54. Mäntylä MV, Adams B, Khomh F, Engström E, Petersen K (2015) On rapid releases and software testing: a case study and a semi-systematic literature review. Empirical Software Engineering 20(5):1384–1425
55. Meszaros G (2007) xUnit test patterns: Refactoring test code. Pearson Education

56. Moha N, Gueheneuc YG, Duchien L, Le Meur AF (2010) Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering 36(1):20–36
57. Moonen L (2001) Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001, p 13
58. Morrison CM (2003) Interpret with caution: multicollinearity in multiple regression of cognitive data. Perceptual and motor skills 97(1):80–82
59. Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on Software engineering, ACM, pp 284–292
60. Nagappan N, Williams L, Vouk M, Osborne J (2005) Early estimation of software quality using in-process testing metrics: a controlled case study. In: ACM SIGSOFT Software Engineering Notes, ACM, vol 30, pp 1–7
61. Nagappan N, Maximilien EM, Bhat T, Williams L (2008) Realizing quality improvement through test driven development: results and experiences of four industrial teams. Empirical Software Engineering 13(3):289–302
62. Nagappan N, Zeller A, Zimmermann T, Herzig K, Murphy B (2010) Change bursts as defect predictors. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering, IEEE, pp 309–318
63. Nelder JA, Wedderburn RW (1972) Generalized linear models. Journal of the Royal Statistical Society: Series A (General) 135(3):370–384
64. Offutt AJ, Untch RH (2001) Mutation 2000: Uniting the orthogonal. In: Mutation testing for the new century, Springer, pp 34–44
65. O'brien RM (2007) A caution regarding rules of thumb for variance inflation factors. Quality & quantity 41(5):673–690
66. Palomba F, Zaidman A (2017) Does refactoring of test smells induce fixing flaky tests? In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 1–12
67. Palomba F, Zaidman A, Oliveto R, De Lucia A (2017) An exploratory study on the relationship between changes and refactoring. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), IEEE, pp 176–185
68. Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R (2017) Toward a smell-aware bug prediction model. IEEE Transactions on Software Engineering 45(2):194–218
69. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018) A large-scale empirical study on the lifecycle of code smell co-occurrences. Information and Software Technology 99:1–10
70. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering 23(3):1188–1221
71. Palomba F, Panichella A, Zaidman A, Oliveto R, De Lucia A (2018) The scent of a smell: An extensive comparison between textual and structural smells. IEEE Transactions on Software Engineering 44(10):977–1000

72. Palomba F, Zaidman A, De Lucia A (2018) Automatic test smell detection using information retrieval techniques. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 311–322
73. Pantiuchina J, Bavota G, Tufano M, Poshyvanyk D (2018) Towards just-in-time refactoring recommenders. In: Proceedings of the 26th Conference on Program Comprehension, pp 312–315
74. Parizi RM, Lee SP, Dabbagh M (2014) Achievements and challenges in state-of-the-art software traceability between test and code artifacts. IEEE Transactions on Reliability 63(4):913–926
75. Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. Journal of Systems and Software 150:22–36
76. Pecorelli F (2019) Test-related factors and post-release defects: an empirical study. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 1235–1237
77. Pecorelli F, Palomba F, Di Nucci D, De Lucia A (2019) Comparing heuristic and machine learning approaches for metric-based code smell detection. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE, pp 93–104
78. Pecorelli F, Palomba F, De Lucia A, Bacchelli A (2020) The relation of test-related factors to software quality: A case study on apache systems - online appendix. URL `https://figshare.com/s/b879d0c891f1423f4935`
79. Pezzè M, Young M (2008) Software testing and analysis: process, principles, and techniques. John Wiley & Sons
80. Qusef A, Bavota G, Oliveto R, Lucia AD, Binkley DW (2014) Recovering test-to-code traceability using slicing and textual analysis. Journal of Systems and Software 88:147–168, DOI 10.1016/j.jss.2013.10.019, URL `https://doi.org/10.1016/j.jss.2013.10.019`
81. Qusef A, Elish MO, Binkley D (2019) An exploratory study of the relationship between software test smells and fault-proneness. IEEE Access 7:139526–139536
82. Rafique Y, Mišić VB (2013) The effects of test-driven development on external quality and productivity: A meta-analysis. IEEE Transactions on Software Engineering 39(6):835–856
83. Rahman F, Devanbu P (2013) How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 432–441
84. Rodríguez-Pérez G, Robles G, González-Barahona JM (2018) Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. Information and Software Technology 99:164–176
85. Seber GA, Lee AJ (2012) Linear regression analysis, vol 329. John Wiley & Sons
86. Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611

87. Shihab E, Jiang ZM, Ibrahim WM, Adams B, Hassan AE (2010) Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp 1–10
88. Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: ACM sigsoft software engineering notes, ACM, vol 30, pp 1–5
89. Spadini D, Aniche M, Bacchelli A (2018) PyDriller: Python Framework for Mining Software Repositories. DOI 10.1145/3236024.3264598
90. Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A (2018) On the relation of test smells to software code quality. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 1–12
91. Spearman C (1904) The proof and measurement of association between two things. American journal of Psychology 15(1):72–101
92. Spinellis D (2005) Tool writing: a forgotten art?(software tools). IEEE Software 22(4):9–11
93. Strecker J, Memon AM (2012) Accounting for defect characteristics in evaluations of testing techniques. ACM Transactions on Software Engineering and Methodology (TOSEM) 21(3):17
94. Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2016) An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp 4–15
95. Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2017) There and back again: Can you compile that snapshot? Journal of Software: Evolution and Process 29(4):e1838
96. Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering 43(11):1063–1088
97. Van Deursen A, Moonen L, Van Den Bergh A, Kok G (2001) Refactoring test code. In: Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001), pp 92–95
98. Van Rompaey B, Demeyer S (2009) Establishing traceability links between unit test cases and units under test. In: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, pp 209–218
99. Vassallo C, Palomba F, Bacchelli A, Gall HC (2018) Continuous code quality: are we (really) doing that? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp 790–795
100. Vassallo C, Panichella S, Palomba F, Proksch S, Gall HC, Zaidman A (2020) How developers engage with static analysis tools in different contexts. Empirical Software Engineering 25(2):1419–1457
101. Wedyan F, Alrmuny D, Bieman JM (2009) The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In: 2009

International Conference on Software Testing Verification and Validation, IEEE, pp 141–150
102. Wohlin C (2014) Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th international conference on evaluation and assessment in software engineering, pp 1–10
103. Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M (2017) How open source projects use static code analysis tools in continuous integration pipelines. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, pp 334–344
104. Zazworka N, Shaw MA, Shull F, Seaman C (2011) Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt, ACM, pp 17–23
105. Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007), IEEE, pp 9–9
106. Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp 91–100

A - List of sources

S01 - Mockus, Audris, Nachiappan Nagappan, and Trung T. Dinh-Trong. "Test coverage and post-verification defects: A multiple case study." 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, 2009.
S02 - Pecorelli, Fabiano. "Test-related factors and post-release defects: an empirical study." Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019.
S03 - Tosun, Ayse, et al. "Predicting defects using test execution logs in an industrial setting." 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, 2017.
S04 - Qusef, Abdallah, Mahmoud O. Elish, and David Binkley. "An Exploratory Study of the Relationship Between Software Test Smells and Fault-Proneness." IEEE Access 7 (2019): 139526-139536.
S05 - Spadini, Davide, et al. "On the relation of test smells to software code quality." 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018.
S06 - Bach, Thomas, et al. "The impact of coverage on bug density in a large industrial software project." 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2017.

S07 - Gren, Lucas, and Vard Antinyan. "On the relation between unit testing and code quality." 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2017.
S08 - Chen, M-H., Michael R. Lyu, and W. Eric Wong. "Effect of code coverage on software reliability measurement." IEEE Transactions on reliability 50.2 (2001): 165-170.
S09 - Del Frate, Fabio, et al. "On the correlation between code coverage and software reliability." Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95. IEEE, 1995.
S10 - Malaiya, Yashwant K., et al. "The relationship between test coverage and reliability." Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering. IEEE, 1994.
S11 - Kochhar, Pavneet Singh, Ferdian Thung, and David Lo. "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems." 2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER). IEEE, 2015.
S12 - Neha, "How To Perform Post-Release Testing Effectively And Minimize Impact Of The Release To Live Clients." https://www.softwaretestinghelp.com/post-release-testing, 2020.
S13 - "Post Release Testing." https://www.professionalqa.com/post-release-testing, 2020.
S14 - Harekal, Divakar, and V. Suma. "Implication of Post Production Defects in Software Industries." International Journal of Computer Applications 975 (2015): 8887.
S15 - Pavneet Singh Kochhar, David Lo, Julia Lawall, Nachiappan Nagappan. Code Coverage and Postrelease Defects: A Large-Scale Study on Open Source Projects. IEEE Transactions on Reliability, Institute of Electrical and Electronics Engineers, 2017, 66 (4), pp.1213 - 1228. 10.1109/TR.2017.2727062. hal-01653728.
S16 - Rajkumar, "Software Test Metrics – Product Metrics & Process Metrics" https://www.softwaretestingmaterial.com/test-metrics/, 2018.
S17 - "What Are Test Metrics?" https://www.sealights.io/agile-testing/test-metrics/, 2020.
S18 - Antinyan, Vard, et al. "Mythical unit test coverage." IEEE Software 35.3 (2018): 73-79.
S19 - King, "Tester's Diary: Getting Ahead With Post-Release Testing" https://blog.gurock.com/post-release-testing/, 2019.
S20 - Hallowell, "Six Sigma Software Metrics, Part 1" https://www.isixsigma.com/tools-templates/software/six-sigma-software-metrics-part-1/, 2020.
S21 - Peters, "Product Managers, do you know how much your bugs cost?" https://deanondelivery.com/product-managers-do-you-know-how-much-your-bugs-cost-72b6e36e7684, 2018.
S22 - Elish, Mahmoud O., and David Rine. "Design structural stability metrics and post-release defect density: An empirical study." 2006 30th Annual Inter-

national Computer Software and Applications Conference (COMPSAC'06 Supplement). IEEE, 2006.
S23 - Vinke, L., P. Klint, and M. Pil. "Estimate the post-release Defect Density based on the Test Level Quality." (2011).
S24 - Rothman, Johanna. "What does it cost you to fix a defect? and why should you care." Retrieved March 13 (2000): 2010.
S25 - Darnell, "Lessons Learned from 2+ Years of Nightly Jepsen Tests" https://www.cockroachlabs.com/blog/jepsen-tests-lessons/, 2019.
S26 - Bach, Thomas, et al. "The impact of coverage on bug density in a large industrial software project." 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2017.
S27 - "How to measure Defect Detection Efficiency/Rate?" https://club.ministryoftesting.com/t/how-to-measure-defect-detection-efficiency-rate/15313, 2018.
S28 - Sridharan, "Testing in Production, the safe way" https://medium.com/@copyconstruct/testing-in-production-the-safe-way-18ca102d0ef1, 2017.
S29 - Sharma, "Why Are Bug Tracking Tools so Important for Testing Teams?" https://dzone.com/articles/why-is-bug-tracking-tool-so-important-for-the-test, 2019.
S30 - "Root Cause Analysis" http://www.helpingtesters.com/root-cause-analysis/, 2017.
S31 - Capgemini, "Capgemini's Quality Blueprint" https://www.capgemini.com/br-pt/wp-content/uploads/sites/8/2017/07/Capgemini___s_Quality_Blueprint.pdf, 2011.
S32 - "Defect Metrics – An indicator of quality of product under test" https://qainfotech.com/defect-metrics-an-indicator-of-quality-of-product-under-test/, 2010.
S33 - "Quality metrics: Defect tracking throughout the software lifecycle" https://searchsoftwarequality.techtarget.com/tip/Quality-metrics-Defect-tracking-throughout-the-software-lifecycle, 2011.
S34 - Cummings-John, "How continuous testing supercharges your development process" https://techbeacon.com/app-dev-testing/use-continuous-testing-supercharge-your-development-process, 2019.
S35 - Riley, "Don't Hate the Tester, Hate the Test Process" https://applitools.com/blog/dont-hate-the-tester-hate-the-test-process/, 2020.
S36 - Starling, "You don't have enough tests and you never will!" https://www.bugsnag.com/blog/better-software-testing, 2017.
S37 - "What to do when defect is found in production but not during the QA phase?" https://sqa.stackexchange.com/questions/27680/what-to-do-when-defect-is-found-in-production-but-not-during-the-qa-phase, 2017.