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

Software Testing and Android Applications: A Large-Scale Empirical Study

Fabiano Pecorelli* · Gemma Catolino · Filomena Ferrucci · Andrea De Lucia · Fabio Palomba

Received: date / Accepted: date

Abstract These days, over three billion users rely on mobile applications (a.k.a. apps) on a daily basis to access high-speed connectivity and all kinds of services it enables, from social to emergency needs. Having high-quality apps is therefore a vital requirement for developers to keep staying on the market and acquire new users. For this reason, the research community has been devising automated strategies to better test these applications. Despite the effort spent so far, most developers write their test cases manually without the adoption of any tool. Nevertheless, we still observe a lack of knowledge on the quality of these manually written tests: an enhanced understanding of this aspect may provide evidence-based findings on the current status of testing in the wild and point out future research directions to better support the daily activities of mobile developers. We perform a large-scale empirical study targeting 1,693 open-source Android apps and aiming at assessing (1) the extent to which these apps are actually tested, (2) how well-designed are the available tests, (3) what is their effectiveness, and (4) how well manual tests can reduce the risk of having defects in production code. In addition, we conduct a focus group with 5 Android testing experts to discuss the findings achieved and gather insights into the next research avenues to undertake. The key results of our study show Android apps are poorly tested and the available tests have low (i) design quality, (ii) effectiveness, and (iii) ability to find defects in production code. Among the various suggestions, testing experts report the need for improved mechanisms to locate potential defects and deal with the complexity of creating tests that effectively exercise the production code.

* Corresponding author: Fabiano Pecorelli, fpecorelli@unisa.it

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

Gemma Catolino
Jheronimus Academy of Data Science & Tilburg University, The Netherlands
E-mail: g.catolino@uvt.nl

Keywords Software Testing · Mobile Applications · Empirical Studies.

1 Introduction

The year 2020 has dramatically changed the way people interact with each other. The rise of social distancing has increased more than ever the need for mobile applications that could connect people and support them when performing any kind of activities [1]; as a matter of fact, these days we have more connected mobile devices (˜7.94 billion) than people [2, 3].

The quality of mobile applications plays a central role for developers to ensure that their apps stay on the market, keep gaining users, and have a high commercial success [4, 5, 6].

Software testing is among the most relevant and well-established methods to control for source code quality [7]. Its relevance is even more critical in mobile computing [8], where continuous releases increase the risk of introducing defects [9, 10]. Furthermore, mobile applications have peculiar characteristics, *e.g.*, apps have multiple sensors and users interact with them through touchscreen, that make testing different and more challenging than those of traditional systems [11]. For the above mentioned reasons, the research community has been actively looking for solutions that could improve the way developers test their applications: these efforts produced the definition of several Graphical User Interface (GUI) testing approaches [5, 12, 13] and frameworks able to ease the verification of both functional and non-functional requirements [14, 15] that can be used to automate some of the developer's activities.

While these approaches have shown to be somewhat actionable, there are a number of limitations, *e.g.*, poor ability to generate valid test data to exercise specific program executions [16], that do not allow the definition of comprehensive, effective, and practical automated testing approach [17]. These limitations make mobile developers reluctant to use automated testing tools and more prone to keep writing tests manually [17, 18, 19].

Unfortunately, the nature of manually written tests has been barely analyzed in literature: empirical studies focused on the characteristics of automated tests [18, 20, 21], while little is known on (1) the extent to which mobile applications contain manual tests, (2) how many of them can be actually executed, (3) what is their quality, considering either test code design and effectiveness metrics, and (4) what is their capabilities in foreseeing defects in production code. An improved understanding of mobile app testing from the perspective of manually written tests may provide important insights to the research community. In fact, should mobile apps be well-tested and/or manually written tests be already effective, the urgency of designing automated approaches could be toned down while focusing on how to complement manually written tests and provide developers with information useful to make tests more effective (*e.g.*, which test data should be used to exercise certain boundary conditions). On the other side, the empirically-grounded results may serve to practitioners as an additional proof of the need for using automatic solutions as well as further supporting the testing research community.

For these reasons, in our previous paper [22] we proposed a large-scale empirical study on the prominence, quality, and effectiveness of the tests manually written by mobile developers. Starting from a dataset composed of 1,693 open-source ANDROID apps, we first extracted manually written test cases and computed how many and which types of tests are actually available as well as which kinds of production classes are more exercised. Secondly, we focused on the design quality of those tests, computing test code quality metrics and smells. Finally, we measured test code coverage and assertion density as proxy metrics to assess test code effectiveness. The study revealed that mobile applications are not sufficiently tested, *e.g.*, we found a median of just 2 test suites per app. Most of the available tests were at unit-level and related to the verification of the application logic, while GUI-related classes and storage of the considered apps were mostly untested. In addition, we discovered that the majority of tests have design issues, as measured by test smells, even though their metric profile would not suggest a low design quality. Finally, both the effectiveness metrics computed were low.

In this paper, we extend the previous work by providing additional insights into the nature of manually written tests of ANDROID apps. More specifically, we deliver the following novel contributions:

1. We conduct an additional analysis into the relation of test cases to the actual defects in production code, with the aim of shedding lights into the practical usefulness of manually written tests with respect to the discovery of issues in production;
2. We include a new qualitative investigation, in which we recruit five ANDROID testing experts and ask them to be part of a focus group [23] aimed at commenting our findings and eliciting what are the current limitations that led to them, in an effort of providing concrete insights into the new research avenues that need to be undertaken by both testing community and tool vendors to better assist developers in their daily activities;
3. We include more metrics when investigating the design aspects of test cases, with the aim of improving the view on the matter. Specifically, we complement the metrics previously taken into account with those available in a comprehensive taxonomy of metrics deemed significant by developers for test code quality [24];
4. We provide a GITHUB repository [25] where we make data, scripts, and transcripts of the focus group publicly available. It provides material that can be used by other researchers to further understand our findings and build upon our work.

Our additional investigation reports that manually written tests have a low ability to foresee defects in production code. Furthermore, the focus group allows us to distill a number of challenges that the research community should face: as an example, testing experts report the need for improved mechanisms to locate production defects in a timely manner and deal with the complexity of creating effective tests.

Structure of the paper. The remainder of the manuscript is as follow. Section 2 presents the research questions driving our study and the dataset exploited. Sections 3 to 7 describe the methodological details and results of the five research questions formulated. In Section 8 we further discuss the main findings achieved, the implications of our study as well as its limitations. Section 9 overviews the related literature, while Section 10 concludes the paper and outlines our future research agenda.

2 Research Questions and Context Selection

The *goal* of the empirical study is twofold: on the one hand, it aims to assess prominence, quality, and effectiveness of test cases written by mobile developers; on the other hand, it aims at identifying the key limitations of mobile testing as perceived by experts when commenting the current status of testing in mobile applications. These two goals have the *purpose* of understanding testing practices, properties, and limitations in the wild, *i.e.*, to what extent mobile apps are tested in practice, what is the outcome of such testing, and what are the factors influencing it. The *perspective* is of both practitioners and researchers: the former are interested in observing how effective are their testing practices, while the latter are interested in understanding whether developers need new instruments to improve the quality of their test suites. In this section, we provide an overview of the research questions driving our empirical investigation and present the dataset employed to address them.

2.1 Research Questions

Our study was structured around five main research questions (**RQ**s). We started by considering some recent findings in the field of mobile software testing [8, 11, 13, 17, 26, 27], which showed that writing tests may be challenging for developers because of (i) the lack of appropriate testing tools and (ii) limited knowledge of testing practices or even willingness of developers to write tests. As such, we first analyzed the prominence of test cases in mobile applications, particularly looking at how many tests are actually developed, which types of tests are implemented and what are the kind of production classes whose functionalities tend to be exercised more. Thus, we asked:

RQ$_1$. *To what extent are test suites developed in mobile apps?*

It is worth noting that, by addressing the first research question, we also provided a larger ecological validity to some preliminary findings [17, 20] on the extent to which mobile apps are tested. After this first analysis, we started a finer-grained investigation of test cases. First, we considered their design, as measured by test code readability and quality metrics [28, 29, 30] and test smells [31, 32, 33]. Second, we took into account the effectiveness of test cases in terms of code coverage [34] and assertion density [35, 36]. Third, we assessed the relation between test cases and post-release defects [37, 38, 39, 40], in an effort of understanding how well can tests prevent the introduction of defects in production code. For these reasons, we defined the following research questions:

RQ$_2$. *What is the design quality of test cases developed in mobile apps?*

RQ$_3$. *What is the effectiveness of test cases developed in mobile apps?*

RQ$_4$. *What is the relation between test cases and post-release defects in mobile apps?*

The analyses of these research questions allowed us to provide a detailed overview of the extent, quality, effectiveness, and fault detection capabilities of tests available in mobile applications. Such an overview was then presented to testing experts with the goal of assessing the quantitative findings against their opinion/expertise and identifying the major reasons behind the current state of testing in mobile applications. This led to our final research question:

RQ$_5$. *What is the developer's take on the current state of mobile apps testing?*

By definition, our methodology followed a mixed-method research approach [41] where the insights derived by mining mobile app data are complemented with qualitative cues coming from experts working on mobile app testing on a daily basis and that can contribute to the development of a comprehensive state of the practice on the matter.

2.2 Context of the Study

The *context* of the empirical study consisted of mobile application data (**RQ$_1$**-**RQ$_4$**) and testing experts (**RQ$_5$**).

As for the former, we considered a set of 1,693 open-source Android apps gathered by mining F-Droid,[1] a repository of free and open-source mobile applications that has been widely employed in the past [16, 42, 43, 44, 45] and that contains a set of applications that enables a good generalizability of the findings with respect to the overall population of free and open-source mobile apps [15, 18, 46, 44]. It is important to note that, while F-Droid contains over

[1] https://f-droid.org

3,000 apps, we narrowed our selection in order to only consider repositories on Github;[2] furthermore, we manually excluded duplicated apps and forks of those already existing in the repository. Based on these filters, we ended up with the final 1,693 open-source mobile apps, whose names, description, and characteristics are reported in our online appendix [25].[3] Our analyses have been conducted on test cases written in Java. We are aware that this is not the only language available to develop tests in mobile applications but it is certainly among the most diffused. In order to reinforce the validity of our study, we sift through our dataset in order to count the number of tests written in Kotlin, a language that has continued to gain adoption in mobile apps testing over the last year [47]. As a result, we found that only 139 out of the 1,693 applications ($< 1\%$) contain at least one Kotlin test. Even if a few applications contain a reasonably high number of Kotlin tests ($\approx$100), we decided not to conduct further analyses, since these just represent rare and isolated cases over the considered dataset.

As for the testing experts, we recruited five professional mobile developers with an Android programming experience ranging between 5 and 10 years. They all work in industry and, on average, they have been developing or contributing as testers to the creation of 25 apps each. While all the participants currently work in industry, two of them still contribute to open-source Android applications, while the other three have worked on open-source applications in the past. The size of the population of developers considered was driven by the specific research approach employed to address $\mathbf{RQ}_5$: focus groups are a form of qualitative research that involves a small number of people sampled conveniently and that can provide expert judgments on the subject of interest [23]. According to well-established guidelines, the ideal size of a focus group is five to eight participants [48]: indeed, larger focus groups are difficult to control and, more importantly, limit each participant's opportunity to share insights and observations [48]. More details on the selection of this research approach as well as on the methodology employed to recruit participants are reported in Section 7.

3 $\mathbf{RQ}_1$ - On the Prominence of Test Cases in Mobile Apps

This section discusses the research methodology and the results achieved when investigating the prominence of test cases in the considered set of mobile apps.

3.1 Research Methodology

To address $\mathbf{RQ}_1$, we first quantified the number of test classes available for each of the apps in our dataset. Starting from their Github repositories, we cloned

[2] https://github.com

[3] With respect to our previous conference paper [22], the number of apps considered decreased from 1,780 to 1,693 because 87 of them were not available anymore at the time of the journal extension.

the apps locally and, afterwards, we performed an exhaustive search through their packages in order to extract classes having *"Test"* as prefix or suffix. As a result of this search process, we computed the number of test classes and methods per app, which corresponds to the number of test suites and test cases available in a mobile application. Furthermore, we proceeded with a more detailed analysis of test suites that aimed at classifying them according to their granularity (*e.g.*, unit vs. integration) and type (*e.g.*, performance). As an automatic classification was not possible, we manually analyzed all the 5,292 extracted test suites using a grounded theory-based methodology [49] which involved two of the authors of this paper (from now on, the *inspectors*). It is worth noting that, in order to exclude false positive tests, during this manual investigation we also checked if the considered classes were actually test classes. No false positives came out from this analysis.

The process consisted of two steps:

Tuning phase. Initially, the inspectors independently classified the same set composed of 500 test suites and annotated in a spreadsheet their granularity and type(s). Whenever possible, the inspectors relied on the available documentation (*e.g.*, code comments) to understand the properties of a certain test: for instance, if developers explicitly stated that the test suite covered the corresponding production class, then the inspectors marked it as a unit test. In the other cases, the inspectors relied on the name of the class as well as analyzed its content to check if (i) only a production class was exercised, *i.e.*, it was a unit test, (ii) more classes were involved to verify the interactions among components, *i.e.*, it was an integration test, or (iii) otherwise, it was a system test. A similar strategy was employed when classifying the type: whenever possible, the inspectors relied on the documentation, while in other cases they manually went over the code to understand which functional or non-functional requirement was exercised. Particularly hard was the case of energy-related tests, for which the inspectors verified whether the test code contained any identifier, API of a third-party library, or profilers of the ANDROID platform connected to energy management. To provide the reader with a concrete example of the classification made, let consider the case of the `ProgramMemoryTest` class of the FINNEYPOKER app. This test suite aims at assessing the memory consumed by the animations implemented in the `Animator` class, which is used by the `PokerActivity`, *i.e.*, the main UI class of the app. As such, the (i) granularity of the test suite was categorized as *'integration'*, since it did not involve one class in isolation nor the system as a whole, and (ii) the type was associated to *'performance'*, as the goal was to assess the consumption of the app in certain conditions.
Through the classification of the same test suites, the inspectors could tune their judgments, find a common way to classify granularity and type of the considered test suites, and discuss their disagreements to better understand the reasoning done by the other inspector. Furthermore, they could compute an initial coding agreement using the Krippendorff's alpha Kr_α [50].

This measured to 0.92, that is considerably higher than the 0.80 standard reference score [51] for Kr_α.

Classification phase. Once completed the tuning phase, the inspectors classified the remaining 4,795 test suites, by analyzing 2,397 and 2,398 each. The outcome allowed the creation of a test suite granularity and type taxonomy for ANDROID apps, which we discussed in Section 3.2.

As an additional analysis aiming at addressing our first research question, we quantified how many and which types of production classes are tested. In this way, we could understand whether developers tend to test only certain specific types of classes (*e.g.*, `Activity` or `Fragment` classes) as well as how much of the production code is covered by a test suite.

To enable this analysis, we first needed to link production to test classes. We relied on the pattern-matching approach designed by Van Rompaey and Demeyer [52]: for each test class, it removes the string *"Test"* from its name and search the production class that matches the remaining part of the name. For instance, using this strategy the test suite *MainActivityTest* would be linked to the production class named *MainActivity*. It is worth mentioning that this linking approach is lightweight in nature and can scale up to the number of apps considered in our study; yet, it has shown similar performance with respect to more sophisticated test-to-code traceability techniques [52].

Afterwards, we computed the number of production classes having a corresponding test suite. As for the type of production classes tested, we performed a first automatic classification, based on keywords, and then we double-checked the classification manually. Specifically, we defined a set of keywords that can distinguish GUI, application logic, and storage components of an ANDROID app. For instance, the GUI keywords included *"activity"* and *"fragment"*, which generally characterize `Activity` and `Fragment` classes used by developers to develop the graphical interface of the app. We included the complete list of keywords used in this stage in our online appendix [25]. Since this automatic classification may be erroneous in some cases, one of the authors of the paper double-checked it and corrected the labels assigned whenever required.

Table 1 Descriptive Statistics of the mobile apps analyzed.

	#Test Suites	**#Test Cases**
Min	0	0
Max	205	2045
Average	3.24	63.30
Median	0	5
Standard Deviation	14.07	202.80
	% Apps Tested	**% Apps Not Tested**
	40	60

3.2 Analysis of the Results

Table 1 reports descriptive statistics on the number of test suites and test cases available in the considered dataset as well as the overall number of mobile applications containing at least one test class. As shown, the first thing that leaps to the eye is that 60% of apps do not present any test case: as such, we can confirm the results obtained by previous work which proved that mobile apps, and in particular ANDROID ones, generally lack tests [18, 20, 21]. This finding reinforces the need for further research on the topic of mobile app testing and, specifically, how to convince developers—who may be non-experienced with the development of source code [53]—of the importance of testing their apps, *e.g.*, by means of empirical evidence showing how lack of testing may worsen the quality of mobile apps. For instance, our results motivate and promote investigations aimed at relating test code quality to change/fault-proneness of the apps [44, 54] or the commercial success of mobile applications [6, 55].

Narrowing our attention to the applications that are actually tested, *i.e.*, the 40% of the apps in our dataset, we computed descriptive statistics related to both test suites and test cases. Table 1 reports the results of this analysis. Looking at the minimum and maximum number of test cases, we found a high variability among the considered applications: indeed, the minimum size of test suites is zero, while it reaches 205 in the best case, with a mean of about three test classes. This result clearly highlights that even apps having Java test suites are in general poorly exercised and would need further support in this activity. The standard deviation value (202.80) confirms the high variability among the considered apps.

Table 2 Granularity and type of test suites developed in the dataset.

Granularity		
Name	Abs.	Rel.
Unit	3,872	73%
Integration	1,273	24%
System	147	3%
Type		
Name	Abs.	Rel.
Functional	4,619	87%
Performance	190	4%
Energy	145	3%
Portability	133	3%
Security	104	2%
Usability	101	1%

As a second part of our analysis aimed at addressing $\mathbf{RQ}_1$, we classified test suites according to their granularity and type. Table 2 summarizes our results. In the first place, we can notice that most of the test suites analyzed are at unit-level: 73% of the tests in our dataset are indeed at this granularity. Interestingly, we discovered that 3,605 of them are directly related to a single production class, while the remaining 268 unit tests exercise more classes at the time. For instance, tests named `IntentTest` or `SwipeTest` indicate generic tests that exercise common functionalities of certain classes without focusing on some of them specifically.

Furthermore, we found that 24% of the test suites pertain to integration testing and aim at exercising how components behave when working together. Finally, a small portion of the considered tests (3%) consists of system tests that aim at testing the application as a whole. Perhaps more interestingly, our investigation into the types of test classes written by developers revealed the existence of a taxonomy composed of six types. As expected, most of the test suites refer to functional tests (87%), namely tests that exercise the input/output of production code classes: this confirms the findings of previous researchers who found that functional testing is the most widely spread type of testing [56, 57]. Subsequently, our categorization shows that performance tests represent the 4% of the available tests: while this number is way lower than the functional tests, this seems to indicate that (1) developers care, even if in a lower extent, of performance of mobile apps, thus confirming previous findings in the field [58, 59] and (2) performance testing is a more delicate problem than for traditional applications [60, 61], suggesting the need of more research to understand better why this happens and what are the consequences.

Furthermore, we found the energy testing is the third more popular type of exercising mobile apps. Also in this case, the number of tests is substantially lower than the one of functional tests; these results are in line with previous findings that highlighted that more automated support to this type of testing would allow developers to better exercise the energy aspects of mobile apps [62, 63]. A small percentage of test suites in our dataset relates to portability testing, namely the types of tests that verify whether the functionalities of an application is compatible with previous versions of the Android operating system [64]: the small amount of tests in this category suggests that more research would be needed in order to understand the reason behind these achievements [65]. Finally, security and usability testing represent the least prominent types of tests in the exploited dataset. On the one hand, the very small amount of tests in these categories clearly highlight that developers are not properly aware of how to cover these aspects [14, 66, 67]: this is particularly worrying in the case of security, also considering the recent data provided by NowSecure[4], which showed that (i) 35% of communications sent by mobile devises are un-encrypted, (ii) 25% of apps have high-risk security flaws, *e.g.*, expose private or sensitive data about a user or their activity, and (iii) 82% of Android devices use an outdated version of the operating system. On the other hand, our findings support and motivate the research done on usability and GUI testing [12, 13], which has been an active field over the last years.

Finally, we focused on the production classes that are actually exercised. Table 3 reports the results achieved: specifically, we split the classes based on their role in the system and, according to this classification, we identified

[4] A well-known security company targeting mobile apps: `https://tinyurl.com/rdhrszc`

Table 3 Types of production classes tested. Abs. = Absolute number; Rel. = Relative number.

Activity		Intent		Fragments		Storage		Application Logic	
Abs.	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**
202	6%	9	1%	52	1%	114	3%	3,228	89%

Table 4 Percentage of tested classes per production class type.

Type	# tests	# production_classes	tests/production_classes %
Activity	202	8,202	2%
Intent	9	38	24%
Fragments	52	5,362	1%
Storage	114	1,713	7%
Application Logic	3,228	14,109	23%

three main categories, namely, *GUI*, *Storage*, and *Application Logic*: the first refers to production classes implementing the logic behind the graphical user interface of mobile apps, the second to the classes that manage the storage of the apps, while the latter to the classes having the single responsibility of implementing business logic of the apps. For the sake of comprehensibility, we split the *GUI* category in the three main class types, namely `Activity`, `Intent`, and `Fragment`. These are the class types that ANDROID developers use to develop the user interface of their applications.

Looking at Table 4, we can observe that most tests in the dataset exercise the application logic of mobile apps. Behind this result, there might be different explanations: First, developers are not properly supported nor aware of current techniques when it comes to the testing of other aspects of their apps [20]. Second, mobile developers are sometimes junior or with less experience than programmers working in other domains and, as shown by previous researchers, they might be less aware of the importance of testing, hence limiting themselves to exercise a limited amount of classes [66].

Finding 1*. Mobile applications contain very few java tests, indeed, only 40% of the apps contain at least one test suite. As for the tested apps, most of the tests pertain to unit tests that exercise the functionalities of the app, while other aspects are not widely considered, like for instance, performance of GUI testing.*

4 RQ$_2$ - On the Design Quality of Test Cases in Mobile Apps

This section reports methodology and results of our analyses to address **RQ$_2$**.

4.1 Research Methodology

Given our original dataset, we had to exclude all the apps without tests from this second research question. This process led us to focus on 673 mobile apps. To assess the design of the considered test cases we covered three macro-aspects that can characterize their maintainability and understandability. Table 5 summarizes the metrics adopted to address $\mathbf{RQ}_2$, while a detailed description is reported in our online appendix [25]. The selection of these metrics was based on the findings reported by Grano *et al.* [24], which presented a taxonomy of metrics deemed significant by developers to measure test code quality. More specifically, such a taxonomy includes behavioral, structural, and execution high-level concerns that developers consider relevant when developing tests and that can be mostly quantified using the metrics in Table 5.

Table 5 List of factors considered in order to measure the design quality of test cases.

Group	Name	Description
Code Metrics	LOC	Number of lines of code of the Test Class
	WMC	Weighted Method Count of the Test Class.
	RFC	Response for a Class.
	IFC	Information Flow Coupling.
	LCOM5	Lack of Cohesion of Test Methods.
	TCC	Tight Class Cohesion.
	LCC	Loose Class Cohesion.
Textual Metrics	Readability	The readability level of the test.
	Comment ratio	Ratio between lines of comments and lines of source code.
Test smells	Eager Test	A test exercising more methods of the production target.
	Indirect Testing	A test interacting with the target via another object.
	Resource Optimism	A test that makes optimistic assumptions on the existence of external resources.
	Mystery Guest	A test that uses external resources (*e.g.*, files or databases).
	Assertion Roulette	A test method containing several assertions with no explanation.

In the first place, we considered test code quality metrics, relying on the metric suite originally defined by Chidamber and Kemerer [28] and other metrics related to code quality. According to Grano *et al.* [24], this set of metrics addresses or helps addressing aspects like scope, size, reusability, and independence of test cases. We computed the Lines of Code (LOC): according to previous achievements [68, 69, 70], having higher size may cause issues for developers with respect to the maintainability of tests as well as to their fault-proneness [71, 72]. For similar reasons, we computed cohesion metrics such as Lack of Cohesion of Test Methods (LCOM5 [73]), Tight Class Cohesion (TCC), and Loose Class Cohesion (LCC) [74]; we measured different metrics as they can provide orthogonal information that may be useful to analyze the cohesion of tests better [75]. Furthermore, we considered the coupling between tests, which is one of the most critical problems when comprehending test code [24, 76]. To this aim, we computed the Information Flow Coupling (IFC), a metric that captures the relations between tests in terms of information exchanged [75] and is among the best suited for assessing the quality of tests [77]. Finally, we considered the complexity of test code. In this case, the rationale comes from previous studies [24, 78, 79, 80] which showed that complexity metrics may be related to both scope and defectiveness of test code as well as may lower the overall understandability of the target of tests [76]. We quantified complexity by computing Weighted Methods per Class (WMC) and Response for a Class (RFC): the former represents the sum of the complexity of the test cases included in a suite, while the latter estimates complexity by considering the number of methods that can potentially be executed in response to a message received by an object of a class. All the metrics were computed at test suite-level, as they can be only extracted at this granularity.

The second set of metrics relate to textual aspects of source code. These can be helpful when quantifying the readability and diagnosability (*i.e.*, the ability of developers to understand faults detected by a test) of test code [24]. First, we computed the overall readability of test cases by relying on the automated approach proposed by Buse and Weimer [81] - we employed the original tool proposed by the authors. Such an approach employs a machine learning-based solution that internally computes 19 metrics covering various aspects of source code that may influence its readability, like the number of keywords or the number of spaces in a piece of code to name a few. The output of the readability tool consists of a readability index ranging between 0 and 1, where 0 indicates an unreadable code and 1 a perfect readability. We also computed the comment ratio, namely the percentage of comments per test method lines of code - the higher this ratio the higher the documentation available for developers and, therefore, the higher its understandability.

Table 6 Descriptive statistics for all metrics considered in RQ_2. Outliers have been removed from distributions.

Metric	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
LOC	2.00	14.00	32.00	46.40	66.00	181.00
WMC	0.00	2.00	4.00	4.80	7.00	17.00
RFC	0.00	6.00	17.00	26.30	39.00	112.00
IFC	0.00	0.19	0.36	0.37	0.53	1.00
LCOM	0.00	0.27	0.50	0.50	0.75	1.00
TCC	0.00	0.00	0.00	0.26	0.50	1.00
LCC	0.00	0.00	0.50	0.50	1.00	1.00
Readability	0.00	0.00	0.00	0.13	0.01	1.00
Comment ratio	0.00	0.00	0.00	0.04	0.03	0.40

To complement the analysis of test code quality metric profiles, we considered test smells, *i.e.*, poor design or implementation choices applied by programmers during the development of test cases [32]. On the one hand, test smells make test code more change- and fault-prone [71] as well as harder to comprehend and maintain [82]. On the other hand, test smells have been shown to be one of the primary causes behind test instability, thus making them extremely harmful for developers [83]. We focused on five forms of test smells widely investigated by the research community, namely *Mystery Guest*, *Resource Optimism*, *Eager Test*, *Assertion Roulette*, and *Indirect Testing.* Their definitions are provided in Table 5. To detect them, we employed the code metrics-based tool developed by Bavota *et al.* [82], which has shown to have high accuracy, close to 86% of F-Measure [82, 84] and has been validated several times in previous work [85, 86, 30, 71], thus making us confident of its suitability for our study. The detector identifies test smells at class-level granularity, which is the same as the other considered metrics. In particular, for each test suite and test smell type, the detector provides a boolean value reporting whether the suite contains at least one instance of the test smell type under investigation.

Table 7 Absolute and relative number of test smells detected. AR = Assertion Roulette; ET = Eager Test; MG = Mystery Guest; RO = Resource Optimism; IT = Indirect Testing.

AR		**ET**		**MG**		**RO**		**IT**	
Abs.	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**	**Abs.**	**Rel.**
2,508	50%	1,556	31%	439	9%	123	3%	371	7%

4.2 Analysis of the Results

Table 6 reports the distributions for all the quality metrics considered in our second research question—note that outliers have been removed from the table for the sake of comprehensibility.

Looking at the table, we first noticed that the LOC metric, which computes the size of test suites, has a median value of 32.00, meaning that the vast majority of the considered tests have a limited size. There are, however, several outliers: we manually analyzed them to better understand how are they composed. From this analysis, we found that all the outliers refer to apps having only one big test class containing several test methods that exercise production code belonging to different classes. As an example, the test `MainActivityTest`, belonging to the package `opencamera.test` of the OpenCamera app, has 12,637 lines of code and implements 1,188 test methods.

When considering complexity metrics like WMC and RFC, our findings suggest that the complexity of tests is generally low (median of 4.00 and 17.00 for WMC and RFC, respectively). The discussion for coupling is more interesting: indeed, the IFC metric has a median of 0.36: this indicates that there exist a non-negligible number of test suites containing methods that depend on other methods of the same class. Besides making such tests less comprehensible [76], this phenomenon may potentially lead to undesired issues like, for instance, potential flakiness due to a test ordering problem, which arises when the execution of a test depends on the execution of another one [87].

Turning the attention to the test case cohesion, we can provide a number of observations. First, the LCOM is almost equally distributed over the spectrum of possible values for this metric. Given its definition, this result indicates that there is a fairly similar amount of test cases that use and not use instance variables defined in the test suite; from a practical perspective, this possibly indicates that the design of tests and their inter-dependence may be affected by the way specific developers implement test cases (*e.g.*, their experience or knowledge of the domain [35, 66]).

The analysis of TCC and LCC provided us with further insights into the cohesion of tests. While the former measures the number of test pairs that directly share instance variables of the test suite, the latter indicates how many of them are either directly or indirectly connected (*i.e.*, share the same variables or are directly connected to the same test). The distribution of those metrics tell us that, while test methods are not always directly connected, they have more often an indirect connection. As such, they follow a similar trend as the one shown in previous studies done on the code quality profile of production classes [88]—there seems to be no peculiarities of these metrics that distinguish tests written by mobile developers.

Finally, we could observe that both the textual metrics considered have a median equals to 0, with a third quartile of 0.01 and 0.03 for readability and comment ratio, respectively. On the one hand, these results suggest that mobile developers spend almost no time addressing documentation concerns in test cases: the low distribution of values for the comment ratio, indeed, reports that only in a few cases developers add comments that explain the responsibilities of the test. On the other hand, the analysis of readability is somewhat even more worrisome: not only tests are not documented, but also potentially hard to comprehend for developers - note that previous literature has shown that poor test readability is often connected to a decrease of bug detection capabilities [29, 89].

As for the test smells, Table 7 reports the distribution of design issues over the considered set of mobile apps. In the first place, we can confirm previous findings in the field [82, 90, 91] and claim that test smells have a high diffuseness also when considering the mobile context. Most of the instances found (50%) refer to *Assertion Roulette*, namely the smell that arises when there are multiple `assert` statements without explanation—this smell lowers understandability and maintainability of test suites [32]. Instances of *Eager Test* are also quite diffused and affect 31% of the test suites in our dataset. According to previous results [71], this smell type is associated with a lower effectiveness of the affected test in terms of fault detection capabilities. The other test smells are less diffused: *Mystery Guest* appears in 9% of the considered tests, while *Resource Optimism* and *Indirect Testing* in 3% and 7%

of the cases, respectively. These percentages are in line with those found in traditional systems by Bavota *et al.* [91] and Grano *et al.* [90], thus indicating that test smells have similar diffusion and relevance in both contexts.

More in general, from our empirical analyses we observed that, while the structural metric profile of tests would not show potential problems affecting their design, the quality of tests is still threatened by the presence of test smells [71]. Despite the fact that they capture two different concepts, this contradiction may potentially indicate that currently available metrics are not enough to measure the actual quality of test suites and, as such, new, different test code metrics that better capture the design quality and understandability of test suites should be further studied and defined.

Finding 2. *The metric profile of the considered test suites does not always indicate the presence of possible issues in test code. However, tests are often affected by test smells that may possibly negatively influence their effectiveness, for instance by leading them to miss faults in production code. Our findings suggest the need for new test code metrics that can better measure the actual quality of test suites.*

5 RQ$_3$ - On the Effectiveness of Test Cases in Mobile Apps

This section details the methodological steps conducted to address our third research question and the results achieved.

5.1 Research Methodology

Test code effectiveness can be estimated in different ways. In the context of our study, we focused on two complementary aspects that have been shown to influence the ability of tests to catch defects in production code, namely line coverage [92] and assertion density [36]. The former measures the amount of code that has been exercised based on the number of Java byte code instructions called by the tests: from a practical perspective, we employed the JaCoCo Android plug-in,[5] a popular code coverage tool, to compute the value for each of the considered test suites. As for the assertion density, this is defined as follow:

$$assertion.density(tc) = \frac{\#assertions(tc)}{LOC(tc)} \quad (1)$$

where tc is the test case under consideration, $\#assertions(tc)$ is the number of assert statements in tc and $LOC(tc)$ is the number of lines of code of the test. Note that we employed the definition of assertion density introduced by

[5] https://github.com/arturdm/jacoco-android-gradle-plugin

Kudrjavets *et al.* [36]. We considered this metrics since it has been associated in the past with a reduction of defect density in production code [35, 36], hence providing an indication of how good a test suite can actually be. To compute assertion density, we developed a tool—available in our online appendix [25].

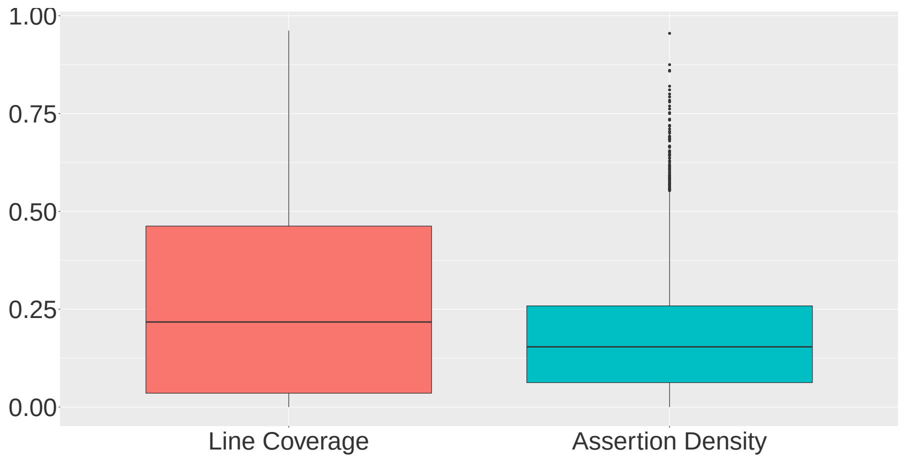

Fig. 1 Distribution of test code quality metrics in our dataset.

5.2 Analysis of the Results

Figure 1 reports the distribution of line coverage and assertion density among all the applications of the dataset. As the figure shows, the values of both the metrics are between 0 and 0.5, excepting for some outliers. The median for line coverage is equal to 0.23, while the one of assertion density is 0.17. We also observe a notable number of outliers, especially when considering the assertion density. Nonetheless, we can claim that these values relate to low effectiveness [93]: their effect on the post-release defects of mobile applications is investigated in the context of $\mathbf{RQ}_4$.

When considering line coverage, the discussion is similar. The vast majority of the test suites have low coverage and cannot properly exercise the corresponding production code. Unlike assertion density, for line coverage we noticed something peculiar and worth of discussion: in some cases, developers discuss about code coverage on the issue trackers and, particularly, on the way they can increase it. For instance, let consider the AnySoftKeyboard app: the developers in this case adopt a pull-based development process where all changes must pass through a pull request before being merged. In most of the cases where new code is committed, developers explicitly ask to the author of the change to verify that the code coverage of unit tests is high enough. As an example, in the issue #551, one developer applied multiple changes to the test code in order to increase its coverage up to 87%. We found similar cases when considering other applications, thus leading us to claim that the developer's perception of code coverage is sometimes pretty high and reflected into the way test cases are developed—this result partially contradicts what reported by Linares-Vásquez *et al.* [19] through their study on the developer's perception of code coverage and indicates that further experiments would be desirable to understand the real value of code coverage for developers. Nevertheless, our findings suggest that mobile programmers still experience troubles when developing effective tests.

Finding 3. *The median code coverage and assertion density are 0.23 and 0.17, respectively. The effect of these low values is analyzed in the next research question. Furthermore, we found that in some cases developers perceive code coverage as highly relevant to accept pull requests.*

6 RQ$_4$ - On the Relation of Test Cases to Post-Release Defects in Mobile Apps

In this section, we describe methodology and results pertaining to **RQ**$_4$, *i.e.*, the relation between test cases and post-release defects of mobile apps.

6.1 Research Methodology

In the context of this research question, we adopted a statistical approach with the aim of relating and assessing how test-related metrics characterizing the goodness of the manually written test cases in mobile applications can indicate the statistical likelihood to have defects in production code. In more practical terms, while in **RQ**$_3$ we measured the effectiveness of test cases only based on metrics computable considering the test code itself, in **RQ**$_4$ we assessed whether and how the goodness of test cases is reflected to the quality of production code, as measured by the number of post-release defects.

The statistical approach employed consisted of multiple steps and methodological choices. The remainder of this section explains our approach in terms of model dependent and independent variables as well as of methods applied to enable valid statistical conclusions and interpretations.

Dependent variable. The dependent variable considered in **RQ**$_4$ is the number of post-release defects, namely the amount of bugs affecting the mobile applications in our study after the snapshot considered for the analysis. The GITHUB repositories pertaining to those apps offer the entire change history in form of commits. We first determined if a commit fixed a defect. In this regard, 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 mobile app's issue tracker system in order to filter only issues related to resolved bugs. Then, we relied on the keyword-matching technique devised by Fischer *et al.* [94] to analyze the commit message and search for the presence of specific keywords, *e.g.*, *'bug'*, *'fix'*, or *'defect'*, that are typically used by developers to mark defect-fixing activities. The application of this technique allowed us to filter out all those commits that referred to issues in the issue tracker but were related to other maintenance and evolution activities. Despite the technique might be considered naive, empirical assessments have shown an accuracy of ≈80% [94, 95] and, for this reason, often been used by previous research [96, 97]. After detecting all defect-fixing commits, we applied the Śliwerski-Zimmerman-Zeller (SZZ) algorithm [98], which is able to identify the defect-inducing commits, namely those modifications that likely introduced defects. The algorithm relies on basic `Git` features such as annotation and blame. Given a defect-fixing commit k as input, SZZ 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 $\textit{rel-fix}_{i,k}$, SZZ extracts the file revision just *before* the defect fixing ($\textit{rel-fix}_{i,k} - 1$).

– Starting from the revision $\textit{rel-fix}_{i,k} - 1$, for each source line in f_i changed to fix the defect k, SZZ identifies 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 so doing, blank lines and lines that only contain comments are identified and excluded using an island grammar parser [99]. This produces, for each production class C_j, a set of $n_{i,k}$ defect-inducing revisions $\textit{rel-defect}_{i,j,k}$, $j = 1 \dots n_{i,k}$. As such, more than one commit can be marked by SZZ as defect-inducing.

Once extracted the defect-inducing commits, we finally computed the post-release defects of a production class as the number of defect-inducing activities involving the class after the release date of the snapshot of the mobile apps considered. From a technical perspective, we employed the SZZ algorithm implemented within PyDriller[6] to compute the dependent variable.

Independent variables. Our aim was to verify the extent to which the goodness of test cases implemented by mobile developers relate to post-release defects. As such, the independent variables of the statistical model comprise the set of metrics considered in $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$. This way we could analyze the impact of various features, *i.e.*, static and dynamic factors, textual metrics as well as test code design, on the ability of tests to act as a guard for the introduction of defects in forthcoming versions of mobile apps.

Confounding factors. Other than to test-related features, the number of post-release defects might be due to additional aspects pertaining to production code quality or the development process. As an example, larger production code classes might be more defect-prone independently from the

[6] Link: https://pydriller.readthedocs.io/.

Table 8 List of confounding factors used in the study.

Group	Name	Description
Structural metrics	PLOC	Number of lines of code of the production class.
	PWMC	Weighted Method Count of the production class.
	PRFC	Response for a production class.
	PIFC	Information Flow Coupling of the production class.
	PLCOM5	Lack of Cohesion of Methods of the production class.
	PTCC	Tight Class Cohesion of the production class.
	PLCC	Lose Class Cohesion 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.
Android-specific smells	Durable Wakelock	A class acquiring a wake-lock without releasing it.
	Inefficient Data Structure	A class that declares a HashMap local variable whose first type argument (i.e., key) is an Integer.
	Internal Setter	A class containing one (or more) non-static method(s) that calls a setter having only a single assignment.
	Leaking Thread	A class exhibiting a Thread that is started but not interrupted.
	Member Ignoring Method	A class containing a non-static and non-empty method that (i) does not access any instance variable; (ii) does not use this and super keywords; (iii) does not override an inherited method.
Development process	Pre-release changes	Number of changes involving the production class before the release date of the considered snapshot.

test cases exercising it. To account for this aspect and avoid a biased interpretation of the results, we computed a set of confounding features that have been shown to influence the defect-proneness of source code. These are summarized in Table 8. As shown, they cover four main characteristics of product and process quality:

– Among the structural metrics, we first take the Production Lines Of Code (PLOC) metric into account. It measures the size of the production classes. Its selection was driven by the fact that PLOC has been often associated to an increase of fault-proneness [68, 69, 70]. To compute it, we employed the automated tool developed by Spinellis [100].

- The Weighted Method per Class (PWMC) and Response for a Class (PRFC) metrics [28] measure the complexity of production code, which is something that naturally influences the defect-proneness of source code [78, 79]. The tool by Spinellis [100] is able to compute this metric too.
- Coupling is another aspect strongly connected to software quality [77]. In our work, we computed the Information Flow Coupling (PIFC) of production classes, *i.e.*, a metric describing the relation between classes in terms of information exchanged.
- Cohesion has also been associated to fault proneness in the past [101]. In this respect, we measured cohesion of production classes by mean of Lack of Cohesion of Methods (PLCOM5), Tight Class Cohesion (PTCC), and Lose Class Cohesion (PLCC).
- Code smells are indicators of sub-optimal design/implementation choices in source code [102]. A number of previous papers have established that those smells heavily increase the chance of production code being faulty [103, 104, 105, 106, 107, 108, 109, 110]. On the one hand, we considered five traditional code smells from the catalog by Fowler [102]. These have different characteristics and cover various program entities, *i.e.*, *God Class*, *Class Data Should Be Private*, *Complex Class*, *Functional Decomposition*, and *Spaghetti Code*. On the other hand, we took the peculiarities of mobile applications into account and computed the so-called Android-specific code smells, *i.e.*, design flaws that are specific of mobile apps and that can impact on defect-proneness in different manners, *e.g.*, by increasing the chance of functional and energy-related defects. In this regard, we computed the 5 code smells mentioned in Table 8.
  As for the actual detection of these code smells, we relied on DECOR [111] for the traditional ones and on aDOCTOR [112] for the Android-specific ones. Both tools have been extensively validated by the research community and showed an excellent accuracy [113, 112], hence representing valid tools to use for our purposes.
- Finally, we computed the number of pre-release changes. This metric captures the quality of the development process [114] and can highlight relevant complementary evolutionary aspects. The metric was computed by mining the change log of the considered apps and counting how often a certain production class has been modified.

Statistical approach. As last step to address our research question, we built a statistical model relating independent and confounding metrics to post-release defects. We opted for the construction of a Generalized Linear Model (GLM) [115]: it models the relationship between a scalar response, like the number of post-release defects, and one or more explanatory variables, *i.e.*, the set of independent and confounding factors, by fitting a linear function whose unknown model parameters are estimated from the data; we used the *'Gaussian'* family when implementing the model. The reason behind the choice of this statistical approach was twofold. At first, it simultaneously

analyzes the effects of both confounding and independent variables on the response variable [116]. Secondly, it does not require the normality of data distribution: in our case, the Shapiro-Wilk normality test [117] rejected the null-hypothesis, hence indicating that our data is not normally distributed. To properly interpreting the statistical results, we accounted for possible issues with multicollinearity [118]. In doing so, we run a hierarchical clustering based on the Spearman's rank coefficient [119] to cluster together variables at different levels of correlation. Afterwards, if two of them had a correlation higher than 0.6, we excluded the more complex one from the model.
Finally, we interpreted the output of GLM by analyzing the statistical significant codes it assigns to each explanatory variable: if a certain metric is statistically significant, this implies that the chances of the effect on the number of post-release defects being random is sufficiently low. We also computed the Adjusted R-squared [120] to assess the goodness of fit of the model, a metric indicating how close the data is to the fitted regression line.

6.2 Analysis of the Results

Table 9 reports the statistical results obtained. Before discussing them, it is worth pointing out that from the total set of variables employed within the model, we had to exclude (i) the variable signaling the presence of the *Functional Decomposition* code smell because it was too correlated with p_lcc, (ii) p_wmc due to its high correlation with p_loc, and (iii) t_wmc and t_rfc, which in this case had high correlation with t_loc. In addition, the statistical model could not consider the *Member Ignoring Method*, *Inefficient Data Structure*, and *Leaking Thread* smells because the Android-specific code smell detector, *i.e.*, ADoctor, did not detect any instance for these smells.

Overall, the model was composed of a total of 24 metrics. The Adjusted R-squared measured 0.114: the value is pretty low, meaning that the input variables, *i.e.*, the set of independent and control factors taken into account, cannot determine well the value of the dependent variable. From a practical perspective, this means that there might exist additional metrics that cover peculiar aspects of mobile applications and that can contribute to the explanation of the statistical power of the model—this seems to suggest the need for more extensive and comprehensive studies on the factors making mobile applications more defect-prone.

Looking at the table, a first aspect to discuss is the high significance of PRE_RELEASE_CHANGES (ρ-value <0.001). The estimate is positive, meaning that the higher the number of changes done before releasing the higher the likelihood that classes will be subject to defects. The result is not really surprising since the past history of a class has been found to strongly influence its future quality in a number of previous work [121, 122]. In this sense, our findings corroborate what discovered in the available literature.

Analyzing the effects of the other confounding factors, we found that (i) the presence of code smells and (ii) the size of production code can influence the

Table 9 Results for **RQ**$_4$ - Factors influencing post-release defects in mobile apps.

	Estimate	S.E.	Sig.
Intercept	-1.18	1.14	
TLOC	-0.01	0.01	
TIFC	1.67	1.11	
TLCC	0.04	0.67	
TTCC	1.91	0.56	***
Assertion Roulette	-0.23	0.53	
Eager Test	-0.34	0.46	
Mystery Guest	1.03	0.81	
Resource Optimism	-2.73	1.37	*
Indirect Testing	1.29	0.63	*
Readability	3.74	2.30	
Comment Ratio	-15.43	7.07	*
Line Coverage	0.31	0.94	
Assertion Density	1.48	1.61	
PLOC	-0.06	0.02	**
PIFC	0.60	1.00	
PLCC	1.94	1.19	
PRFC	0.33	0.99	
PTCC	-0.51	0.71	
God Class	2.51	1.32	.
Class Data Should Be Private	6.91	5.25	
Complex Class	11.49	4.32	**
Durable Wakelock	1.00	4.30	
Internal Setter	2.84	3.51	
Pre-release Changes	0.78	0.05	***

Multiple R-squared: 0.169; Adjusted R-squared: 0.114

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

number of POST_RELEASE_DEFECTS. As for the former, our results still confirm that the presence of design issues in production code, and of GOD CLASS and COMPLEX CLASS instances in particular, leads the affected classes to be more defect-prone [105, 123]. As for the latter, it is worth noticing that the estimate is slightly negative (-0.06): this indicates that the lower the number of production code lines, the higher the number of post release defects. While at first glance this could sound counter-intuitive, there are two observations to do. In this first place, the estimate is close to zero and, therefore, there might not be evident reasons making the metric connected to the dependent variable. In the second place, however, the result could be explained by a larger adoption of third-party libraries [44] that has the effect of sensibly reducing the amount of code in the app but, at the same time, increasing the likelihood of developers introducing defects because of API misinterpretation or the usage of defect-prone APIs—as shown in literature [54].

Turning the attention to our core interest, namely the impact of tests on POST_RELEASE_DEFECTS in mobile applications, we found that cohesion of test cases is a very relevant aspect (ρ-value<0.001) that influences the dependent variable with a positive estimate, *i.e.*, the higher the cohesion the higher the number of defects. Also in this case, the result is counter-intuitive. The findings of $\mathbf{RQ}_1$ have shown that mobile applications are characterized by a few number of tests with a limited size: having a high cohesion in these cases may indicate that the test exercises only few and strongly cohesive functionalities in production code, hence neglecting others. The low coverage observed in $\mathbf{RQ}_3$ seems to confirm that the few tests available are not able to verify the production code in an appropriate manner.

The significance of test smells confirm previous results achieved in the context of traditional applications [40, 71]. Interestingly, we noticed that the INDIRECT TESTING smell has an estimate of 1.29, hence directly affecting the number of post-release defects. This smell arises when a test exercises multiple classes of the production code, not being able to focus on a specific target class. As previously shown [71], the lack of focus is among the key test-related problems increasing the defect-proneness of production code.

Last but not least, the COMMENT RATIO of test cases was found to be statistically significant, with an estimate of -15.43. This means that the lower the amount of documentation in tests the higher the number of defects in production. Our results in $\mathbf{RQ}_2$ revealed that test cases of the considered applications are indeed poorly documented: the statistical analysis suggests that such a lack of documentation represents a serious threat to the reliability of source code. This is in line with previous findings [124] showing that non-commented tests cause a decrease of program comprehensibility and, for this reason, developers might encounter difficulties in detecting defects in production code.

Finding 4. *Post-release defects are mainly influenced by the number of changes performed on production code. A few test-related-factors negatively contribute to the phenomenon as well. The cohesion of test cases is one of the most significant factors influencing the number of post-release defects. Other aspects such as the comment ratio and presence of test smells are still important but with a lower significance.*

7 $\mathbf{RQ}_5$ - On the Developer's Opinions on Mobile App Testing

This section reports on methodology and results that address $\mathbf{RQ}_5$.

7.1 Research Methodology

The analyses done so far provided a quantitative view on the state of the practice in mobile application testing. While we could provide some additional insights through our manual investigation of the considered apps, these are clearly not generalizable and would require further investigation. Being aware of that, our last research question sought to elicit the opinions of developers having a solid experience in the context of mobile app testing.

Previous work in the field have exploited survey research to complement mining studies (*e.g.*, [6, 9]): by nature, surveys provide quantifiable data that can be used to establish, on a large scale, the maturity of a technology or, in the software engineering field, of novel instruments. Yet, surveys are not interactive and the data coming from them are likely to lack details or depth on the topic being investigated [125]. As the goal of $\mathbf{RQ}_5$ was to gather insights and in-depth opinions on the findings achieved in the mining study, we then preferred not to go for a survey, favoring a more qualitative approach like the one of focus groups, which are rarely large enough to draw definitive conclusions, but have the advantage of fostering discussions and uncovering ideas that otherwise would have been missed [23, 125].

More specifically, focus group research is defined as a small group of carefully selected participants who contribute to open discussions for research [23]. In the context of our study, a focus group enables a joint discussion on current testing practices and limitations among the recruited experts. From a methodological standpoint, the participants were invited to join an online Zoom meeting[7]—note that at the time of the study we were not allowed to run a physical meeting because of the COVID19 pandemic.

The first two authors of the paper acted as moderators. At the beginning of the meeting, a 2-minute presentation of the participants was allowed, so that all of them got to know the others. Then, the first author of the paper provided an overview of the main goals of the study, a brief explanation of the research methods employed, and a detailed discussion of the results achieved in the context of $\mathbf{RQ}_1$-$\mathbf{RQ}_4$. To this purpose, a 10-minute slideshow was prepared: the last slide was designed to contain a summary of the main findings of the study and was kept shared with the participants till the end to allow them to always bear in mind the achieved results. It is worth noting that, while summarizing the results, the first author highlighted that these came out exclusively from open-source applications—this was done with the aim of setting the participants' expectations on the open-source side. The presentation provided to the participants is available in our online appendix [25].

In the second part of the focus group, participants were asked to comment on the results and report experiences with respect to the testing of mobile applications that might explain the quantitative results of the study. This part of the meeting lasted 45 minutes and was kept by the moderators highly interactive: they did not simply leave the word to each participant, but asked others to comment and reflect on the possible reasons behind what s/he was reporting. The entire discussion was recorded and stored for analysis.

Upon completion of the recording, Zoom provided as output both the video registration and a text file reporting the transcript of the meeting. The first

[7] `https://www.zoom.us/en/`

two authors of the paper reviewed the transcript together in order to identify the main insights and comments left by the participants. We finally addressed $\mathbf{RQ}_5$ by reporting the most relevant insights from the focus group.

Table 10 Background of the focus group participants.

ID	Dev. Exp.	Mobile Exp.	Working context
P1	13	10	Software Corporation
P2	11	6	Airline company
P3	11	6	Mobile Software House
P4	10	6	Researcher & Spin-off CTO
P5	11	6	Testing researcher

7.2 Analysis of the Results

Table 10 reports the background of the five experts to the focus group. As shown, all of them have a similar development experience and, since at least five years actively work on the development of mobile applications. Three participants (P1, P2, and P3) have a full-time appointment in large software companies of different nature: P1 works within a well-known US software and technology corporation founded in 1975, P2 is employed in a Dutch airline company, while P3 works for a multinational corporation that develops mobile applications. The fourth participant (P4) is a Senior Research Associated in an Italian university, but also has a partial appointment as Chief Technology Officer of a technological spin-off operating in the context of big data analytics. Finally, P5 is a researcher in the field of software testing having, however, experience in the development of mobile applications, *i.e.*, P5 was employed within a mobile software company before starting the academic career.

For the sake of readability, in the following we report the key insights coming from the focus group by discussing each research question independently.

Commenting $\mathbf{RQ}_1$. After the introductory part, the two moderators started the focus group by asking whether the presented results were in line with the participant's expectations of how mobile testing is applied in practice. All participants agreed on the fact that, based on their experience, mobile apps are poorly tested and that, unfortunately, our results for $\mathbf{RQ}_1$ provide a representative overview of what happens in practice. In this respect, P1 commented that *"the majority of mobile developers have poor testing skills and, indeed, they mostly perform manual testing by adding pre- and post-conditions in the production code"*. In other words, according to P1's opinion, developers tend not to develop test cases but verify the behavior of their apps by crafting specific statements within the production code to verify pre- and post-conditions while developing or evolving the code. It also turned out that these statements might be even opportunistically disabled, *i.e.*, test code is activated for debugging purposes and then commented when the app is finally released.

The other participants agreed with the P1's take, yet they also pointed out that this is a typical behavior observed with small applications developed by a few number of developers having low or no software engineering skills. P4 also found another motivation for the low amount of tests: in some cases, s/he said, *"testing specific usage scenarios is challenging, since mobile apps can interact with various hardware components and sensors (like the Bluetooth)"*. Hence, the overall discussion not only highlighted education challenges, *e.g.*, how to address the problem of testing mobile applications at an education level, but also that developers sometimes need specific test beds or mocking strategies to simulate the behavior of external hardware components.

When it comes to the classification of test cases reported in $\mathbf{RQ}_1$ (see Table 2), the participants unanimously agreed that the higher percentage of unit tests discovered in our study is due to the higher simplicity of performing unit testing with respect to the other types. Furthermore, P1 pointed out that *"the lower amount of integration and system tests might be also explained with developers writing tests that cover very specific, domain- and context-dependent use cases of their applications"*. Reasoning around this statement, it seems that the small number of integration and system tests may not necessarily be a problem in practice, since developers might have a deeper knowledge of the use cases that users will more frequently apply. Perhaps more importantly, P2 and P5 highlighted that mobile developers have the opportunity to test apps as a whole by employing established behavioral testing framework like Cucumber[8] and others—which are actually widely used in practice [11]. Interestingly, however, P2 reported that: *"behavioral tests provide developers with the perception that everything is working properly, but there must still be uncaught bugs. Nevertheless, in most cases developers accept those bugs since the cost of writing integration and system tests would be excessive"*. These observations allowed us to provide two main conclusions. On the one hand, having a few integration and system tests might not necessarily be an issue as long as they are complemented with testing frameworks that can exercise the app in different manner. On the other hand, our findings further support the work done by the research community around automatic test case generation: as a matter of fact, writing tests remain a costly activity that should be appropriately supported with automated mechanisms, especially in a context where continuous releases are expected to be delivered.

Our participants also actively discussed the results reporting non-functional attributes to be poorly tested. P3 argued that *"while there exist some frameworks to assist developers while exercising, for instance, performance and energy constraints, software companies do not often care about these types of testing"*, *i.e.*, companies focus on functional requirements, neglecting non-functional ones. Moreover, P3 explained that *"it is hard to create non-functional tests because the definition of oracles is challenging and developers do not often have expertise to deal with the complexity of these tests"*. The other participants also pointed out a lack of tools able to *measure*

[8] `https://cucumber.io`

non-functional aspects. To draw a conclusion, the discussion raised two main challenges for researchers: the need for more research on oracles and the need for more techniques/tools able to properly assess non-functional attributes of mobile applications - especially when these depend on external events.

Commenting RQ$_2$. Moving the attention to the developer's take on the results achieved when considering the design quality of the manual tests object of our investigation, the participants were quite interested in commenting on the documentation of those tests. P2 pointed out that *"is it not really surprising that amount of comments is low. There is a growing trend in industry for which developers should not spend too much time in documenting test cases since they will not frequently change over time"*. P5 confirmed this line of thinking and added that *"in the company I worked, the governance used to employ a strict naming convention to enforce developers write test names that clearly define the goals of the test and its target; in this way, failing tests could be easily retrieved and diagnosed"*. These observations led us to first argue that the documentation strategies for test cases are drastically different from those of production code, *i.e.*, the focus is on names that can quickly evoke the responsibilities of the tests rather that on code comments that are more costly to write and may possibly become outdated. At the same time, we still see room for better assisting developers by means of improved automatic code comment generators as well as refactoring instruments that can update tests and their documentation when new changes to production code are applied.

When addressing the readability of test code, participants were instead reluctant to consider this as a relevant aspect for design quality. In this respect, P2 reported that *"the readability of tests is much lower than the one of production code, but this is quite normal given the different goal that testing has"*. In other words, the participant argued that the main objective of test cases is to find defects, independently from how good the test code is readable. Additionally, P4 claimed that *"the readability values are really low considering that tests are usually short pieces of code. Perhaps, the metric employed does not appropriately capture the readability of tests"*. This statement led to the formulation of a hypothesis: the currently available test metrics do not properly measure the desirable properties of test cases. We further investigated such a hypotheses in the remainder of the discussion.

The participants also had concerns when discussing the structural code metrics. P3 reported that *"in my experience, low cohesion and high coupling in tests indicate that there is something wrong in production code"*; the other participants agreed with this statement and confirmed that the status of test cases often simply reflects the quality of production code. Going deeper into the discussion of the specific metrics, P2 added that *"the coupling values are particularly worrisome, it is likely that tests only exercise a few methods of the production code"*, while P1 reported that *"the test cohesion must not necessarily be high, yet this may indicate that there exist poorly cohesive test suites exercising more production classes"*. Based on these observations, it seems clear that there is a strong relation between production and test quality and, therefore, our results can reflect the more general poor quality of mobile applications—hence corroborating the findings by Linares-Vásquez *et al.* [126].

To conclude the discussion on **RQ**$_2$, the moderators explicitly asked participants whether the available test code metrics are actually suitable to provide developers with relevant information about the design quality of tests. In response, P2 explained that *"there are some good metrics, like LOC, while others are quite meaningless in practice, like the comment ratio"*. More in general, P1 reported that *"the level and goal of the metrics that we expect to assess tests is different from those of production code"*. Although our findings in this respect are not conclusive, we believe they raise the need for further investigations into the real usefulness of the current metrics.

Commenting RQ$_3$**.** The participants went through the results achieved when computing code coverage and assertion density. In the first place, all participants agreed that the results are not surprising: these are, indeed, in line with their expectations since *"it is a standard, yet unhealthy practice that of considering test cases as second-class citizens"*, said P5.

P2 added that *"most of the tests analyzed are at unit-level, which makes the low coverage even more worrisome: they are likely to exercise only the easiest parts of the production code"*. More in general, P3 commented that *"even if the coverage is objectively low, this metric does not necessarily imply that the tests cannot catch defects"*.On the one hand, this confirms recent findings by Grano *et al.* [24], *i.e.*, test case effectiveness is a multifaceted concept that should be assessed by combining multiple metrics. On the other hand, the participants' opinions led us to further push the need for additional investigations into the definition of novel metrics to assess test code quality and effectiveness.

The discussion on assertion density led to a similar conclusion. P1 reported that *"some tests might be useful even when they have no assertions; the assertion density is a metric, but not necessarily good"*. In addition, P1 and P4 observed that the metric computation is naturally biased by the amount of code required to setup the test environment, *i.e.*, if a test requires more code to prepare the environment the denominator will be higher, leading to a lower density that does not necessarily indicate the poor quality of the test.

Concluding the discussion for these results, we could confirm that the general feeling of our participants was that the available testing metrics are not enough to provide a comprehensive view of test code effectiveness.

Commenting RQ$_4$**.** When discussing the results on the relation between test code properties and post-release defects, participants basically confirmed the opinions given for the previous research questions. In the first place, they pointed out that test cases of such a poor quality cannot provide any significant indication to developers and are, naturally, going to fail in catching defects in production code. They also clarified how the continuous changing nature of mobile apps make the testing development process particularly challenging, since tests must be frequently updated and, as a matter of fact, there is typically not enough workforce, time nor experience to write effective tests and deal with the intrinsic complexity given by the environmental constraints (*e.g.*, hardware components and sensors).

Finding 5. *The quantitative results of our study reflect the expectations that developers have of the status of mobile app testing. The participants provided further explanations and insights into the matter,* e.g.*, by raising specific education and technical challenges that the research community should carefully look at. In addition, our focus group let emerge the need for additional/novel metrics able to better measure both quality and effectiveness of test cases.*

8 Discussion, Implications, and Limitations

In this section, we present the main insights coming from the results of the study and the limitations that might have affected the validity of the study.

8.1 Discussion and implications

The results of our empirical study provided a number of insights and practical implications for the research community that need further discussion.

Mobile apps contain very few numbers of Java tests. The first evident, worrisome result of our study clearly indicated the lack of tests in mobile applications: not only the mean number of tests is ≈ 3, but also the percentage of apps without any test is rather high (60%). There are multiple factors possibly contributing to this finding. First, our dataset is composed of open-source mobile applications that can be developed under different conditions with respect to other applications: as an example, they can be developed by inexperienced or novice programmers with little knowledge on testing practices [53]. During the focus group discussion, our participants also raised this point and commented on how the lack of software engineering/testing expertise can have a significant impact on how mobile applications are tested. At the same time, the developers involved in $\mathbf{RQ}_5$ reported that test cases are typically seen as second-class citizens, especially in a dynamic environment like the one of mobile development, where a continuous release model might soon make tests outdated other than increasing the cost required to maintain and evolve them. In this respect, our findings corroborate previous results obtained by Beller *et al.* [127] on the lack of developer's willingness in successfully evolving tests. As an exemplary case appearing in our dataset, let consider the case of Acastus Photon,[9] an online address/POI search for navigation apps. Looking deeper at its issue tracker and the developer's comments, we noticed that the developers of the

[9] https://f-droid.org/en/packages/name.gdr.acastus_photon/

app have consciously postponed some testing activities with the aim of entering the market faster or because of the lack of time to dedicate to testing. For instance, in one of the issues still open on the issue tracker (#2), one of the core developers of the app posted the following comment:

"[...] I'm probably going to merge the build changes later on too. [...] I don't have time to test them right now so just merging master."

As shown, in this case the developer decided not to test the newly committed code change because of the need to other modifications to the production code. Even without an extensive search, we found similar cases in other apps of our dataset. Finally, our findings can be also due to the limited automated support that developers have when testing their apps. As pointed out in the context of our focus group, mobile developers experience very specific challenges when developing test cases, like the need for considering external events coming from hardware components or sensors. By looking at the state of the art, there exist a number of tools to automate GUI testing (*e.g.*, MONKEY or SAPIENZ [12]), other than some frameworks for behavioral-driven testing (*e.g.*, the CUCUMBER tool mentioned in $\mathbf{RQ}_5$), yet only a few automated and practical mechanisms are available for the generation of functional and non-functional test cases (*e.g.*, EVOSUITE [128]). In addition, these tools do not explicitly take into account the problem of mocking hardware components. As such, our findings do not only support the research in the field of automatic test case generation, but also call for the definition of mobile-specific instruments.

🔑 Developers typically see test cases as second-class citizens, thus postponing testing activities in favor of other aspects perceived as more important (e.g., time-to-market). Mobile-specific automatic test case generation techniques are needed to support developers during testing activities.

On integration and system testing. According to our findings, most of the test suites present in mobile apps pertain to unit testing, while we discovered only a limited amount of them referring to integration and system testing [129]. This result might be due to various reasons. While the lack of automated tools and/or support mechanisms might influence the way mobile developers verify their apps for integration- and system-level faults, it is also worth remarking that different verification mechanisms, like manual validation, crowd-testing [130], or even the use of external tools that can exercise the apps to verify certain specific properties (*e.g.*, energy consumption [15, 131]), might be put in place. As an example, our study highlighted that, depending on the context, the lack of automated integration and system JAVA tests might not necessarily be a problem because developers may want to test the use cases that users perform more often when using the app, hence leading to the application of non-systematic or opportunistic testing strategies that are not automated, e.g., crowd-testing [130]. On the one hand, our findings may possibly pave the way for novel smart techniques that can analyze runtime usage logs to recommend when to add test cases or suggest modifications to the current ones. On the other hand, we point out the need for additional investigations into the methodologies employed by developers to perform integration and system testing.

Test cases in mobile applications are mainly developed at unit-level granularity. This could be possibly due to the adoption of other types of testing techniques for integration- and system-level, such as automatic or crowd testing.

Enabling testing of non-functional attributes. Most of the tests developed in mobile apps relate to functional aspects of production code, while few of them refer to testing of non-functional attributes like, for instance, energy consumption, security, or performance. The developers involved in our focus group clearly pointed out how hard the development of these tests can be. There are two key limitations of the state of the art in this respect. In the first place, defining an oracle for these types of test is a challenging or even a non-deterministic task, *e.g.*, the oracle of an energy test must necessarily take into account the non-determinism of energy measurements. In the second place, our study highlighted a worrisome lack of instruments that support developers when measuring non-functional aspects: these have been often connected to the commercial success of mobile applications [54, 43, 6], making the lack of testing a threat for the overall sustainability of these apps. On the basis of these results, we therefore argue that more methods to manage the complexity of non-functional attribute testing should be designed: while the research community has been working already on the measurement side (*e.g.*, [15]), to the best of our knowledge there is no study targeting the oracle problem when considering non-functional testing.

The great majority of tests developed in mobile applications relate to functional aspects rather than non-functional ones. More methods that support testing of non-functional attributes are needed.

The design quality of mobile apps is low. Our findings report that most of the tests analyzed are affected by some form of test smells. Previous researches have shown how these problems can turn into critical threats to the effectiveness of tests [40, 71]. To identify them, some test smell detectors have been developed in the past [132, 133, 84] and experimented in the context of mobile applications [134, 135]. Yet, there is no empirically-assessed technique available to automatically refactor test code. As such, the practical support provided to mobile developers is still very limited.

Test cases in mobile apps are characterized by a low design quality. Mobile developers need to be supported by the definition of novel techniques for automatically identifying and refactor design issues.

On the need of novel metrics for test code quality. When analyzing the quality of test suites, we also computed code metrics capturing cohesion, coupling, complexity, and documentation aspects. Our quantitative analyses

($\mathbf{RQ}_2$) revealed a contradiction between the metric profile of tests and the actual presence of design issues. While the values of the metrics would not indicate problems with the design of test cases, we discovered that test smells are often present and lower the maintainability and understandability of tests. By contrasting these results with those achieved in our qualitative investigation ($\mathbf{RQ}_5$), we discovered that the meaningfulness of these metrics is limited. The involved developers not only presented practical scenarios where the metrics could not be relevant (*e.g.*, companies may implement guidelines for naming conventions, discouraging developers to write code comments), but also pointed out that the level and scope of test metrics are different from those of production code. Our findings indicate that researchers should go beyond the currently available code metrics and define novel indicators that can better quantify the quality of test cases. Furthermore, on the basis of our results we argue that both available and prospective quality metrics should be re-contextualized for mobile applications, for instance by providing an easy way to quantity how much the quality of tests depend on the quality of production code.

Current test metrics seem to be not very related to the actual presence of design issues. Therefore, researchers should spend more effort in defining novel indicators having higher explanatory power for test case quality.

On the effectiveness of test suites. Another relevant finding is the low effectiveness of the test cases analyzed when considering both code coverage and assertion density. On the one hand, these metrics have been previously positively correlated to the fault detection capabilities of tests [40]: as such, their low value is somewhat worrisome and depict a critical situation for the mobile applications considered. On the other hand, however, developers raised some practical critiques to those metrics: the assertion density computation might be biased by the amount of code required to setup the test environment, while the coverage only provides a part of the whole story. These observations further confirm the need to establish novel methods to evaluate test cases. Moreover, the results corroborate recent findings showing that the effectiveness of tests represent a phenomenon that goes way beyond the currently available methods [24]: as such, we argue that newly proposed metrics should consider the aspects deemed important for developers and, more importantly, possibly be directly assessed against the developer's perception of test code effectiveness.

Test cases in mobile applications have low effectiveness. While one could immediately think that this represents a criticality for mobile application, it could be also possible that the metrics on which our considerations are based cannot fully explain test effectiveness. Also in this case, the definition of new metrics could help for further and deeper analyses.

Teaching software testing. To some extent, both the quantitative and qualitative findings of our study showed that mobile developers do not have proper testing skills. This is even worse when considering the constraints that mobile apps might have, *e.g.*, the interaction with sensors. As a consequence, our study highlights the need for interventions on an educational level. This concerns both software engineering courses covering basic testing skills, and more sectoral courses on mobile development providing students with specific mobile-oriented skills: we not only argue that the focus on testing practices should increase from a technical perspective, but educators should further pushed on the value of having quality and effective test suites in practice, for instance by showing the extent to which testing is connected to the failure of software engineering projects [136] or by designing additional seminars on the matter, trying to engage students with practitioners.

Mobile developers do not have proper testing skills. Educators should consider giving more importance to design quality and effectiveness of test cases rather than only focusing on the technical perspective.

8.2 Threats to Validity

Some possible limitations could have biased our findings; this section discusses how we mitigated them.

Threats to construct validity. This category refers to the relationship between theory and observation. A first point of discussion concerns the dataset of mobile apps exploited in the study. Previous work has found that some of the applications available in the F-Droid repository are very basic projects [137, 138], thus possibly biasing the conclusions of empirical studies. To overcome this limitation, we manually went over each of the initially downloaded apps in order to discard those that appeared to be too trivial to be considered. In particular, we looked at their repository in order to check whether they result active, *e.g.*, in terms of commits, conjecturing that trivial apps are not updated and actively developed, *e.g.*, since could be part of a university project for an exam.

In all our research questions, we relied on some automatic tools for various reasons. In $\mathbf{RQ}_1$ we employed the test-to-code traceability approach defined by Van Rompaey and Demeyer [52] to find the production classes exercised by the considered tests, as well as we used a keyword-based tool to classify production classes according to their role in the system. While the test-to-code traceability approach has been validated several times in the past showing a very high accuracy, we manually double-checked the classifications done by our own keyword-based tool in order to fix them whenever needed. In $\mathbf{RQ}_2$, we employed an automatic test smell detector: its accuracy has been previously assessed [82, 84] showing to be close to 86% in terms of F-Measure. Such an accuracy makes us confident of the reliability of the instrument and its suitability for the purpose of the study. Nonetheless, it is worth pointing out that we employed the test smell detector at test suite-level, *i.e.*, it outputs a boolean value indicating the presence of at least one instance of the various test smell types considered: additional analyses, conducted at a finer granularity, would be desirable for corroborating our results. Finally, in **RQ**$_3$ we exploited JaCoCO and our own tool to compute code coverage and assertion density, respectively. The former has been widely used in the past by the research community and, therefore, can be considered as de-facto standard. The latter is a simple tool computing the ratio between assertions and test class KLOC, that we tested before exploiting it in our study. For the sake of verifiability, we also made the tool available in our appendix [25].

Threats to conclusion validity. Limitations of this type concern the relationship between experimentation and outcome. In principle, our study should be considered as an evidence-based experiment in which our observations and findings come from the analysis of the actual evidences left by mobile developers with respect to their testing activities. The large-scale nature of the study allows us to provide the research community with results and findings having a large ecological validity; in addition, we also contrasted the quantitative findings with the results obtained from a qualitative study conducted in the form of focus group. Such a mixed-method approach allowed us to not only discuss overall findings from a large set of mobile apps, but also to describe the main underlying reasons under them.

The metrics used to address our **RQ**s are all well-established in the research community and allowed us to overview the status of mobile testing in a comprehensive manner. However, it is worth discussing that, in **RQ**$_3$, we estimated test code effectiveness by looking at line coverage and assertion density, without considering another well-known indicator such as the mutation score [139], *i.e.*, the amount of artificially created production faults that a test can detect. We are aware that this metric could have provided an additional view of the effectiveness of tests, but unfortunately all the available mutation tools (*e.g.*, Pit[10]) do not scale up to the size of our empirical study and, therefore, the computation of mutation coverage would have been prohibitively expensive. Nevertheless, it is also worth remarking that Gopinath *et al.* [140] reported line coverage to be the coverage-criterion that is more related to test case effectiveness and, perhaps more importantly, it has a direct relation with mutation operators that act at line-level. Given such a relation, we are confident that the results discussed in the paper would have not drastically changed if mutation coverage would have been included in our empirical study. At the same time, we recognize that replications targeting this aspect would be beneficial to provide a further view on the matter.

In the context of **RQ**$_1$, we performed a manual analysis to classify granularity and type of tests in our dataset. To this aim, we followed a grounded-theory approach [49] where two authors first classified an identical set of tests in order to tune their judgment and proceeded with the classification process smoothly. Of course, we still cannot exclude the presence of some imprecision in the classification, however the high agreement reached by the inspectors makes us confident of the reliability of the process conducted.

[10] https://pitest.org

In **RQ**$_4$, we designed a statistical model to relate test code metrics to post-release defects. In this respect, we controlled test-related factors for possible confounding effects due to the characteristics of production code, considering both product and process metrics as well as Android-specific code smells, that have been shown to be connected with to post-release defects. Another threat is related to the actual suitability of the employed statistical method, *i.e.*, Generalized Linear Model. It is worth recalling that, before selecting it, we verified the assumptions that the model makes on the underlying data. Nonetheless, 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 [115].

As a final remark, our **RQ**$_5$ has been designed to follow a focus group methodology. This is a qualitative research method that, by nature, does not require the participation of a large amount of experts — it is explicitly designed to have a small group of participants able to foster discussion and provide insights/recommendations on the phenomenon of interest [23, 125]. The conclusions reached might not be definitive: we are aware of that, yet we preferred to have in-depth opinions on the findings achieved in the mining study rather than more generic results that might have achieved using other instruments, *e.g.*, surveys. As usual, however, replications are desirable and might provide complementary insights into the testing of mobile applications.

Threats to external validity. As for the generalizability of the results, our study targeted a large set of open-source applications, thus allowing the verification of the characteristics of tests on a large scale. Nevertheless, it is worth pointing out that our findings may differ in different contexts, *e.g.*, in closed-source apps testing practice results different, as well as settings, *e.g.*, when considering test smells other than those taken into account. As such, further replications of our study would be desirable and are already part of our future research agenda.

9 Related Work

The ever increasing complexity of mobile applications, given by their peculiarities (*e.g.*, ensuring that the application is downloadable, works seamlessly, and gives the same experience across various devices and users) as well as by their differences with respect to standard applications [53, 141], has pushed the research community to define methods to support developers with testing activities [8]. Researchers have been investigating how developers test their mobile applications in comparison to standard systems [8], showing dissimilarities, peculiarity and possible effective practices. Given the goals of our paper, in this section we mainly focus on the studies aiming at analyzing testing practices of mobile developers by (i) surveying and/or interviewing practitioners [19, 18] and (ii) performing mining software repository studies [20, 18].

Linares-Vásquez *et al.* [19] surveyed 102 open-source ANDROID developers on their habits when performing testing, focusing on (i) their practices and preferences, (ii) automated testing methods employed, and (iii) perception of code coverage as indicator of test code quality. As a result, they found that developers rely on usage models (*e.g.*, use cases, user stories) of their applications when designing test cases and perceive code coverage not necessarily important for measuring the quality of test cases. Subsequently, the same authors [11] investigated current tools and frameworks that support mobile testing practices, including benefits and trade-offs between different approaches/tools. A similar work has been done by Choudhary *et al.* [16], which benchmarked automated test input generation tools, discovering that MONKEY, the random testing tool integrated within ANDROID STUDIO is still among the best ones.

Along the same direction, the work of Kochhar *et al.* [18] surveyed 83 ANDROID developers and 27 WINDOWS app developers at MICROSOFT to study techniques, tools, and types of testing used in the mobile context. At the same time, they also analyzed 600 ANDROID apps in terms of the extent to which they are tested, assessing line and block coverage. The results showed that ANDROID apps are not properly tested (*i.e.*, 86% do not present any test cases), and this seems to be in line with the perception of developers, who are not aware of many existing testing tools.

Erfani *et al.* [17] interviewed 191 mobile developers asking about current testing practices. Results showed that there is a lack of robust monitoring, analysis, and testing tools. The work of Silva *et al.* [21] showed similar results. Indeed, they studied 25 open-source ANDROID apps in terms of test frameworks adopted, highlighting that mobile apps are not properly tested; a possible reason behind this result may be related to the lack of effective tools [21]. A recent study by Cruz *et al.* [20] investigated working habits and challenges when testing mobile apps. In particular, they analyzed 1,000 ANDROID apps, showing that testing technologies (*e.g.*, JUNIT) are absent in the 60% of the cases; however, when a mobile application is tested, the authors observed an increment of contributors and commit, moreover they noticed that mobile apps with tests have got an high number of minor code issues. Finally, the most recent work was performed by Lin *et al.* [142]; in particular, they conducted a large-scale analysis, over 12.000 mobile apps, to understand how test automation works in this context, *i.e.*, tendency to write tests and practice itself. Moreover, they analyzed how test automation impacted the popularity and surveyed 148 developers to have feedback about automation test adoption.

With respect to the papers discussed above, our work can be seen as complementary. In the first place, *our study is more focused on the analysis of both the presence and quality of tests of mobile apps rather than the usage of the testing tool and the perception of developers[142]*: thus, we analyze the actual tests written by mobile developers and not their perception with respect to testing practices. In the second place, we provided a larger ecological validity to some previous studies which investigated how much mobile apps are tested [18, 20]: our dataset was indeed composed of over 1,600 mobile applications, which allowed us to provide more concrete conclusions than previous studies. Third, our study included a large-scale analysis of the design quality of test cases and considers both test smells and code quality metrics, which represent a key novelty of our work. Finally, when comparing our work with the one by Kochhar *et al.* [18], it is important to point out that we analyzed the effectiveness of tests by not only considering traditional coverage indicators, but also taking into account assertion density, which has been shown to impact the ability of tests to find defects in production classes [36, 35].

10 Conclusion

In this paper, we conducted a large-scale investigation into the characteristics of test suites written by developers of mobile applications under four perspectives, namely (1) whether and to what extent these apps are tested and which kind of tests are developed, (2) what is the design quality of the test suites, in terms of code metrics and test smells, (3) what is the effectiveness of tests, considering assertion density and code coverage, and (4) how test-related metrics are associated to the defect-proneness of production code. The quantitative insights coming from the analysis of these aspects were then discussed in the context of a focus group involving 5 mobile testing experts, who commented the achieved results and provided practical explanations and experience reports useful for understanding the status of testing in mobile as well as the key limitations that must be addressed.

The main results of the study highlight that 40% of the considered apps have at least one test suite; developers mostly test source code to exercise its functionalities, while other types of testing are less widespread. Test smells represent a key problem for most of the test suites, since some of them exhibit characteristics making them possibly flaky. Their effectiveness is low when considering all the computed metrics. Finally, the characteristics of test cases lead to a negative impact on production code and, indeed, most of the statistically significant test-related factors in our study are correlated to a higher defect-proneness of the corresponding classes. These findings reflected the expectations that the involved experts had when thinking of the status of mobile testing. Furthermore, from the focus group discussion we could delineate a number of education and technical challenges that future research should address. To sum up, our paper made the following contributions:

1. A large-scale empirical study on the prominence, design quality, effectiveness, and capabilities of test cases manually written by developers in the context of mobile applications;
2. Insights coming from a focus group composed of five testing experts that highlighted critical aspects to further consider that the research on mobile testing should further consider;
3. An online appendix [25] containing data and scripts used to conduct our study, and which can be used to further understand our findings and build upon our work.

Our future research agenda follows the roadmap defined in Section 8.1 and includes the definition of new test metrics as well as new techniques that can automate some of the testing processes of mobile developers. At the same time, we aim at studying the aspects treated in this paper on an even larger scale, by considering applications coming from different domains and contexts (*e.g.*, closed-source apps), test cases written in other languages (*e.g.*, Kotlin), as well as datasets containing open-source mobile applications that are also available on the Google Play Store—this would increase the generalizability of the findings reported in our paper when considering datasets collected cross-listed set of apps across both Google Play and GitHub. Finally, we also plan to investigate additional test-related and mobile-specific metrics that can be correlate with the presence of defects in production code.

Acknowledgment

Gemma is partially supported by the European Commission grant no. 825040 (RADON). Fabio gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Projects No. PZ00P2_186090 (TED). The authors would like to thank the Associate Editor and the anonymous Reviewers for their insightful comments provided during the peer-review process, which were instrumental to improve the quality of the manuscript.

References

1. N. Y. Times. (2020, Mar.) How covid19 has changed social interactions. https://www.nytimes.com/interactive/2020/04/07/technology/coronavirus-internet-use.html.
2. Statista. (2020, Mar.) Number of smartphone users worldwide. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.
3. B. of Apps. There are 12 million mobile developers worldwide, and nearly half develop for android first. https://goo.gl/RNCSHC.
4. W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE transactions on software engineering*, vol. 43, no. 9, pp. 817–847, 2016.
5. A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* ACM, 2013, pp. 224–234.
6. F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "Crowdsourcing user reviews to support the evolution of mobile apps," *Journal of Systems and Software*, vol. 137, pp. 143–162, 2018.
7. G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* John Wiley & Sons, 2011.
8. H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *Proceedings of*

the 7th International Workshop on Automation of Software Test. IEEE Press, 2012, pp. 29–35.

9. M. Nayebi, B. Adams, and G. Ruhe, “Release practices for mobile apps–what do users and developers think?” in *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)*, vol. 1. IEEE, 2016, pp. 552–562.
10. S. McIlroy, N. Ali, and A. E. Hassan, “Fresh apps: an empirical study of frequently-updated mobile apps in the google play store,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1346–1370, 2016.
11. M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, “Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing,” in *2017 IEEE International Conference on Software Maintenance and Evolution ICSME*. IEEE, 2017, pp. 399–410.
12. K. Mao, M. Harman, and Y. Jia, “Sapienz: Multi-objective automated testing for android applications,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.
13. G. Grano, A. Ciurumelea, S. Panichella, F. Palomba, and H. C. Gall, “Exploring the integration of user feedback in automated testing of android applications,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 72–83.
14. J. Gao, W.-T. Tsai, R. Paul, X. Bai, and T. Uehara, “Mobile testing-as-a-service (mtaas)–infrastructures, issues, solutions and needs,” in *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE, 2014, pp. 158–167.
15. D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, “Software-based energy profiling of android apps: Simple, efficient and reliable?” in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 103–114.
16. S. R. Choudhary, A. Gorla, and A. Orso, “Automated test input generation for android: Are we there yet?” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering ASE*. IEEE, 2015, pp. 429–440.
17. M. E. Joorabchi, A. Mesbah, and P. Kruchten, “Real challenges in mobile app development,” in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 15–24.
18. P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, “Understanding the test automation culture of app developers,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation ICST*. IEEE, 2015, pp. 1–10.
19. M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, “How do developers test android applications?” in *2017 IEEE International Conference on Software Maintenance and Evolution ICSME*. IEEE, 2017, pp. 613–622.

20. L. Cruz, R. Abreu, and D. Lo, "To the attention of mobile software developers: guess what, test your app!" *Empirical Software Engineering*, pp. 1–31, 2019.
21. D. B. Silva, A. T. Endo, M. M. Eler, and V. H. Durelli, "An analysis of automated tests for mobile android applications," in *2016 XLII Latin American Computing Conference CLEI.* IEEE, 2016, pp. 1–9.
22. F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba, "Testing of mobile applications in the wild: A large-scale empirical study on android apps," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 296–307.
23. S. Wilkinson, "Focus group methodology: a review," *International journal of social research methodology*, vol. 1, no. 3, pp. 181–203, 1998.
24. G. Grano, C. De Iaco, F. Palomba, and H. C. Gall, "Pizza versus pinsa: On the perception and measurability of unit test code quality," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2020, pp. 336–347.
25. F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba, "Software testing and android applications: A large-scale empirical study — online appendix," https://github.com/SeSa-Lab/OnlineAppendices/tree/main/EMSE21-MobileApps, 2021.
26. G. Catolino, D. Di Nucci, and F. Ferrucci, "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, 2019, pp. 99–110.
27. R. Minelli and M. Lanza, "Software analytics for mobile applications–insights & lessons learned," in *2013 17th European Conference on Software Maintenance and Reengineering.* IEEE, 2013, pp. 144–153.
28. S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.
29. G. Grano, S. Scalabrino, R. Oliveto, and H. Gall, "An empirical investigation on the readability of manual and generated test cases," in *Proceedings of the 26th International Conference on Program Comprehension*, 2018.
30. F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ACM, 2016, pp. 130–141.
31. G. Meszaros, *xUnit test patterns: Refactoring test code.* Pearson Education, 2007.
32. A. Van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, 2001, pp. 92–95.
33. M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference*

on Automated Software Engineering, 2016, pp. 4–15.
34. R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 72–82.
35. G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "How the experience of development teams relates to assertion density of test classes," in *2019 IEEE 35th International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2019, p. to appear.
36. G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the relationship between software assertions and faults: An empirical investigation," in *2006 17th International Symposium on Software Reliability Engineering.* IEEE, 2006, pp. 204–212.
37. M.-H. Chen, M. R. Lyu, and W. E. Wong, "Effect of code coverage on software reliability measurement," *IEEE Transactions on reliability*, vol. 50, no. 2, pp. 165–170, 2001.
38. N. Nagappan, L. Williams, M. Vouk, and J. Osborne, "Early estimation of software quality using in-process testing metrics: a controlled case study," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.
39. N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Software Engineering*, vol. 13, no. 3, pp. 289–302, 2008.
40. F. Pecorelli, F. Palomba, and A. De Lucia, "The relation of test-related factors to software quality: A case study on apache systems," *Empirical Software Engineering*, vol. xxx, no. xxx, p. xxx, 2020.
41. J. W. Creswell, "Mixed-method research: Introduction and application," in *Handbook of educational policy.* Elsevier, 1999, pp. 455–472.
42. I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE software*, vol. 31, no. 2, pp. 78–86, 2013.
43. F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *Proceedings of the 39th international conference on software engineering.* IEEE Press, 2017, pp. 106–117.
44. P. Salza, F. Palomba, D. Di Nucci, A. De Lucia, and F. Ferrucci, "Third-party libraries in mobile apps," *Empirical Software Engineering*, pp. 1–37, 2019.
45. H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2014.
46. D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith, "A dataset of open-source android applications," in *Proceedings of the 12th Working Conference on Mining Software Repositories.* IEEE Press, 2015, pp. 522–525.

47. B. G. Mateus and M. Martinez, “An empirical study on quality of android applications written in kotlin language,” *Empirical Software Engineering*, vol. 24, no. 6, pp. 3356–3393, 2019.
48. M. Cleary, J. Horsfall, and M. Hayter, “Data collection and sampling in qualitative research: does size matter?” *Journal of advanced nursing*, pp. 473–475, 2014.
49. N. Pidgeon and K. Henwood, *Grounded theory.* na, 2004.
50. K. Krippendorff, *Content analysis: An introduction to its methodology.* Sage publications, 2018.
51. J.-Y. Antoine, J. Villaneau, and A. Lefeuvre, “Weighted krippendorff’s alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation.” 2014.
52. B. Van Rompaey and S. Demeyer, “Establishing traceability links between unit test cases and units under test,” in *2009 13th European Conference on Software Maintenance and Reengineering.* IEEE, 2009, pp. 209–218.
53. T. Wasserman, “Software engineering issues for mobile application development,” 2010.
54. G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, “The impact of api change-and fault-proneness on the user ratings of android apps,” *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2014.
55. G. Catolino, “Does source code quality reflect the ratings of apps?” in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems.* ACM, 2018, pp. 43–44.
56. M. Beller, G. Gousios, A. Panichella, and A. Zaidman, “When, how, and why developers (do not) test in their ides,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 179–190. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786843
57. G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, “Not all bugs are the same: Understanding, characterizing, and classifying bug types,” *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019.
58. H. Kim, B. Choi, and W. E. Wong, “Performance testing of mobile applications at the unit test level,” in *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement.* IEEE, 2009, pp. 171–180.
59. T. Das, M. Di Penta, and I. Malavolta, “A quantitative and qualitative investigation of performance-related commits in android apps,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2016, pp. 443–447.
60. M. Nagappan and E. Shihab, “Future trends in software engineering research for mobile apps,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 21–32.

61. C. Laaber and P. Leitner, “An evaluation of open-source software microbenchmark suites for continuous performance assessment,” in *Proceedings of the 15th International Conference on Mining Software Repositories.* ACM, 2018, pp. 119–130.
62. R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, “Anti-patterns and the energy efficiency of android applications,” *arXiv preprint arXiv:1610.05711*, 2016.
63. F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, “On the impact of code smells on the energy consumption of mobile applications,” *Information and Software Technology*, vol. 105, pp. 43–55, 2019.
64. L. Wei, Y. Liu, and S.-C. Cheung, “Taming android fragmentation: Characterizing and detecting compatibility issues for android apps,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2016, pp. 226–237.
65. A. Mesbah and M. R. Prasad, “Automated cross-browser compatibility testing,” in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011, pp. 561–570.
66. R. Pham, S. Kiesling, O. Liskin, L. Singer, and K. Schneider, “Enablers, inhibitors, and perceptions of testing in novice software teams,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 30–40.
67. P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, “Vision: automated security validation of mobile apps at app markets,” in *Proceedings of the second international workshop on Mobile cloud computing and services.* ACM, 2011, pp. 21–26.
68. A. G. Koru, D. Zhang, K. El Emam, and H. Liu, “An investigation into the functional form of the size-defect relationship for software modules,” *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 293–304, 2009.
69. F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, “An exploratory study on the relationship between changes and refactoring,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC).* IEEE, 2017, pp. 176–185.
70. N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, “Investigating the impact of design debt on software quality,” in *Proceedings of the 2nd Workshop on Managing Technical Debt.* ACM, 2011, pp. 17–23.
71. D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, “On the relation of test smells to software code quality,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2018, pp. 1–12.
72. A. Vahabzadeh, A. M. Fard, and A. Mesbah, “An empirical study of bugs in test code,” in *2015 IEEE international conference on software maintenance and evolution (ICSME).* IEEE, 2015, pp. 101–110.
73. B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, “Coupling and cohesion (towards a valid metrics suite for object-oriented analysis

and design),” *Object oriented systems*, vol. 3, no. 3, pp. 143–158, 1996.
74. S. Counsell, S. Swift, and J. Crampton, “The interpretation and utility of three cohesion metrics for object-oriented design,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 2, pp. 123–149, 2006.
75. B. Ujhazi, R. Ferenc, D. Poshyvanyk, and T. Gyimothy, “New conceptual coupling and cohesion metrics for object-oriented systems,” in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation.* IEEE, 2010, pp. 33–42.
76. C. S. Yu, C. Treude, and M. Aniche, “Comprehending test code: An empirical study,” p. to appear, 2019.
77. E. Fregnan, T. Baum, F. Palomba, and A. Bacchelli, “A survey on software coupling relations and tools,” *Information and Software Technology*, 2018.
78. D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, “A developer centered bug prediction model,” *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2018.
79. N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, “Change bursts as defect predictors,” in *2010 IEEE 21st International Symposium on Software Reliability Engineering.* IEEE, 2010, pp. 309–318.
80. T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007).* IEEE, 2007, pp. 9–9.
81. R. P. Buse and W. R. Weimer, “Learning a metric for code readability,” *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.
82. G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, “Are test smells really harmful? an empirical study,” *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.
83. M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, “Understanding flaky tests: The developer’s perspective,” p. to appear, 2019.
84. F. Palomba, A. Zaidman, and A. De Lucia, “Automatic test smell detection using information retrieval techniques,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2018, pp. 311–322.
85. V. Garousi and B. Küçük, “Smells in software test code: A survey of knowledge in industry and academia,” *Journal of systems and software*, vol. 138, pp. 52–81, 2018.
86. F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, “On the diffusion of test smells in automatically generated test code: An empirical study,” in *Proceedings of the 9th International Workshop on Search-Based Software Testing.* ACM, 2016, pp. 5–14.
87. Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *Proceedings of the 22nd ACM SIGSOFT International*

Symposium on Foundations of Software Engineering. ACM, 2014, pp. 643–653.
88. L. H. Etzkorn, S. E. Gholston, J. L. Fortune, C. E. Stein, D. Utley, P. A. Farrington, and G. W. Cox, "A comparison of cohesion metrics for object-oriented systems," *Information and Software Technology*, vol. 46, no. 10, pp. 677–687, 2004.
89. J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 831–841.
90. G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019.
91. G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* IEEE, 2012, pp. 56–65.
92. Y. Wei, B. Meyer, and M. Oriol, "Is branch coverage a good measure of testing effectiveness?" in *Empirical Software Engineering and Verification.* Springer, 2012, pp. 194–212.
93. B. Marick *et al.*, "How to misuse code coverage," in *Proceedings of the 16th Interational Conference on Testing Computer Software*, 1999, pp. 16–18.
94. M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on.* IEEE, 2003, pp. 23–32.
95. F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 977–1000, 2018.
96. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.
97. S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.
98. J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.
99. L. Moonen, "Generating robust parsers using island grammars," in *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, 2001, p. 13.
100. D. Spinellis, "Tool writing: a forgotten art?(software tools)," *IEEE Software*, vol. 22, no. 4, pp. 9–11, 2005.

101. V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.
102. M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
103. M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 23–31.
104. T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 33, 2014.
105. F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.
106. F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Transactions on Software Engineering*, 2017.
107. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.
108. ——, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.
109. F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," 2019.
110. M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.
111. N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.
112. E. Iannone, F. Pecorelli, D. Di Nucci, F. Palomba, and A. De Lucia, "Refactoring android-specific energy smells: A plugin for android studio," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 451–455.
113. F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.
114. A. E. Hassan, "Predicting faults using the complexity of code changes," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 78–88.

115. J. A. Nelder and R. W. Wedderburn, "Generalized linear models," *Journal of the Royal Statistical Society: Series A (General)*, vol. 135, no. 3, pp. 370–384, 1972.
116. U. Halekoh, S. Højsgaard, J. Yan *et al.*, "The r package geepack for generalized estimating equations," *Journal of Statistical Software*, vol. 15, no. 2, pp. 1–11, 2006.
117. S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.
118. R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, no. 5, pp. 673–690, 2007.
119. C. Spearman, "The proof and measurement of association between two things," *American journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904.
120. N. R. Draper and H. Smith, *Applied regression analysis.* John Wiley & Sons, 2014, vol. 326.
121. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.
122. S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 2007, pp. 489–498.
123. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, pp. 1–34, 2017.
124. S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 547–558.
125. P. H. Rossi, J. D. Wright, and A. B. Anderson, *Handbook of survey research.* Academic Press, 2013.
126. M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc, "Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 232–243.
127. M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 261–284, 2017.
128. G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025179

129. G. Balogh, T. Gergely, Á. Beszédes, and T. Gyimóthy, "Are my unit tests in the right package?" in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 137–146.
130. N. Leicht, I. Blohm, and J. M. Leimeister, "Leveraging the power of the crowd for software testing," *IEEE Software*, vol. 34, no. 2, pp. 62–69, 2017.
131. A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer: A hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 12–21.
132. B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.
133. M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *Software Testing, Verification and Validation (ICST)*, 2013, pp. 322–331.
134. A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: an exploratory study." in *CASCON*, 2019, pp. 193–202.
135. A. Peruma, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "An exploratory study on the refactoring of unit test files in android applications," in *Conference on Software Engineering Workshops (ICSEW'20)*, 2020.
136. D. A. Tamburri, F. Palomba, and R. Kazman, "Success and failure in software engineering: A followup systematic literature review," *IEEE Transactions on Engineering Management*, 2020.
137. F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli, "A graph-based dataset of commit history of real-world android apps," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 30–33.
138. F.-X. Geiger and I. Malavolta, "Datasets of android applications: a literature review," *arXiv preprint arXiv:1809.10069*, 2018.
139. Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.
140. R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 854–874, 2017.
141. J. Zhang, S. Sagar, and E. Shihab, "The evolution of mobile apps: An exploratory study," in *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile*. ACM, 2013, pp. 1–8.
142. J.-W. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source android apps: A large-scale empirical study," in *2020 35th IEEE/ACM International Conference on Automated Software Engineer-*

ing (ASE). IEEE, 2020, pp. 1078–1089.