Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps

Fabiano Pecorelli
fpecorelli@unisa.it
SeSa Lab
University of Salerno, Italy

Gemma Catolino
gcatolino@unisa.it
SeSa Lab
University of Salerno, Italy

Filomena Ferrucci
fferrucci@unisa.it
SeSa Lab
University of Salerno, Italy

Andrea De Lucia
adelucia@unisa.it
SeSa Lab
University of Salerno, Italy

Fabio Palomba
fpalomba@unisa.it
SeSa Lab
University of Salerno, Italy

ABSTRACT

Nowadays, mobile applications (a.k.a., apps) are used by over two billion users for every type of need, including social and emergency connectivity. Their pervasiveness in today's world has inspired the software testing research community in devising approaches to allow developers to better test their apps and improve the quality of the tests being developed. In spite of this research effort, we still notice a lack of empirical studies aiming at assessing the actual quality of test cases developed by mobile developers: this perspective could provide evidence-based findings on the current status of testing in the wild as well as on the future research directions in the field. As such, we performed a large-scale empirical study targeting 1,780 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, and (3) what is their effectiveness. The key results of our study show that mobile developers still tend not to properly test their apps. Furthermore, we discovered that the test cases of the considered apps have a low (i) design quality, both in terms of test code metrics and test smells, and (ii) effectiveness when considering code coverage as well as assertion density.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*; *Empirical software validation*;

KEYWORDS

Mobile apps; Software testing; Empirical software engineering.

ACM Reference Format:
Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2018. Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps. In *ICPC 2020: IEEE/ACM International Conference on Program Comprehension, May 23–24, 2020 - Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The usage of mobile devices such as smartphones and tablets is playing a central role in everyday life [51]. This is also reflected in the growth of the app industry: more than 5 million mobile apps are available in popular marketplaces like Apple App Store and Google Play Store, for a total of over 205 billion downloads and 12 million mobile developers maintaining them [65, 82]. In such a context, developing and delivering high-quality apps represents a strict requirement for developers to stay on the market, keep gaining users, and maintain a high commercial success [48, 51, 68].

Software testing is one of the most relevant and well-established methods to control for source code quality [61] and to enable the understandability of the functionalities being implemented in a system [80]. This activity is even more pressing in the context of mobile computing [60], where continuous releases of an app have the high risk of introducing defects and decrease software reliability [52, 64]. At the same time, the unique characteristics of mobile apps, *e.g.*, the fact that users can interact via touch-screen and a number of sensors, present brand new challenges for developers when testing the source code of their applications [46].

All these reasons have led the research community to focus on testing best practices and techniques that could better support mobile developers [51]; these concern several papers on Graphical User Interface (GUI) testing [30, 48, 49] as well as on the definition of suitable frameworks to ease testing activities [19, 23]. However, despite the effort spent so far, the solutions available still present limitations that prevent the realization of a comprehensive, effective, and practical automated testing approach [14, 36]. As a consequence, mobile developers still mostly conduct testing activities manually [36, 39]. Looking at the available literature, we notice that this aspect has been barely treated so far: while many researchers focused on whether developers use tools for automating testing activities [16, 39, 79], very few of them have analyzed how mobile applications are actually tested [39], what is the percentage of executable tests within mobile apps and, more importantly, their quality in terms of code design metrics and effectiveness (*e.g.*, coverage).

Investigating mobile app testing from the perspective of manually written tests may provide important insights to the research

ICPC 2020, May 23–24, 2020, Seoul, South Korea

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

community. Indeed, 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 contrary, 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 the reasons above, in this paper we propose a large-scale empirical study on the prominence, quality, and effectiveness of the tests manually written by mobile developers. Particularly, we consider a dataset of 1,780 open-source Android applications and extract test cases written by developers with the first aim of understanding how many and which types of tests are actually available as well as which kinds of production classes are more exercised. In the second step, we focus on the design of these tests, intended as test code quality metrics and smells. Finally, we measure the effectiveness of mobile test cases, considering two widely-used metrics such as code coverage and assertion density.

Our empirical study clearly reports that mobile applications are not sufficiently tested and, indeed, we find a median of just 2 test suites per app. Furthermore, these tests are mostly at unit-level and concern with the application logic, while GUI-related classes and storage of the considered apps remain mostly untested. As for the test code design, we discovered that most of them are affected by some form of test smells, despite having a metric profile that would not suggest the low design quality. Finally, all the effectiveness metrics measured are low, meaning that tests are likely to have a poor fault detection capability. The results of our paper allow us to delineate a number of open challenges in the field of mobile testing, especially related to the need for automated techniques able to assist developers when writing and maintaining tests, but also for scaling testing approaches at integration and system level and easing the applicability of non-functional testing.

Structure of the paper. The remaining of the manuscript is as follow. Section 2 presents the research questions driving our study and the dataset exploited. Sections 3, 4, and 5 describe the methodological details and results of the three research questions formulated. In Section 6 we further discuss the main findings achieved, the implications of our study as well as its limitations. Section 7 overviews the related literature, while Section 8 concludes the paper and outlines our future research agenda.

2 RESEARCH QUESTIONS AND CONTEXT SELECTION

The *goal* of the empirical study is to assess prominence, quality, and effectiveness of test cases written by mobile developers, with the *purpose* of understanding testing practices and properties in the wild, *i.e.*, to what extent mobile apps are tested in practice and what is the outcome of such testing. 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 what are the instruments that mobile developers would need 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 some relevant information on the dataset employed to address them.

2.1 Research Questions

Our study was structured around three main research questions (**RQ**s). We started by considering some recent findings in the field of mobile software testing [10, 30, 36, 46, 56, 60], 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 by developers, 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 the context of mobile apps?*

It is worth noting that, by addressing the first research question, we also provided a larger ecological validity to some preliminary findings [16, 36] on the extent to which mobile apps are tested. After this first analysis, we started a finer-grained investigation of test cases. On the one hand, we considered their design, as measured by test code quality metrics [13, 69] and test smells [55, 85, 88]. On the other hand, we took into account the effectiveness of test cases in terms of code coverage [29] and assertion density [11, 43]. 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?*

The analyses of these research questions allowed us to provide a detailed overview of the extent, quality, and effectiveness of tests available in mobile applications, characteristics that have been shown to have a strong impact on the ability of tests to detect faults in production code [35, 72, 81].

2.2 Context of the Study

The *context* of the empirical study consisted of 1,780 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 [14, 37, 58, 70, 78] 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 [19, 39, 42, 78]. It is important to note that, while F-Droid contains over 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] https://f-droid.org
[2] https://github.com

1,780 open-source mobile apps, whose names and characteristics are reported in our online appendix [8].

3 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 **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 the apps locally and, afterwards, we performed an exhaustive search through their packages in order to extract classes having *"Test"* as prefix or suffix—note that we did not limit the search to classes explicitly using the JUNIT framework since some categories of tests (*e.g.*, UI test cases) may be not developed using it.

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 these tests 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 [77] which involved two of the authors of this paper (from now on, the *inspectors*). 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 documentation available (*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. 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 *'energy'*, 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 [41]. This measured to 0.92, that is considerably higher than the 0.80 standard reference score [1] 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 [89]: 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 [89].

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. For the sake of space limitations, we included the complete list of keywords used in this stage in our online appendix [8]. 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	294	3587
Average	7.26	52.30
Median	2	4
Standard Deviation	20.19	211.83
	% Apps Tested	**% Apps Not Tested**
	41	59

3.2 Analysis of the Results

Table 1 reports descriptive statistics related to 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 59% 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 of tests [16, 39, 79]. 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 [91]—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 [3, 78] or the commercial success of mobile applications [9, 68]. As an example, let consider the case of QuickDic, an app that generates dictionary files that can be used offline; this app has 3,587 lines of code and 662 test cases. Currently, QuickDic has a rate of 4.5 stars on the Google Play Store and users reviewed it as a very useful and fast app. Moreover, developers seem to be very active when replying to the problems found by users, thus suggesting that a higher attention to the quality of mobile apps is rewarded with a higher commercial success [68, 70]. Conversely, looking at the Tweetings for Twitter app, which is a basic Twitter client having 349 lines of code and no tests, we noticed that it has 2.5 stars and users often complain about the fact that the app is very slow and has several problems with push notification; this could potentially reflect the absence of appropriate testing activities.

Narrowing our attention to the applications that are actually tested, *i.e.*, the 41% 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 294 in the best case, with a median of just two test classes. This result clearly highlights that even apps having test suites are in general poorly exercised and would need further support in this activity. The standard deviation value (211.83) confirms the high variability among the considered apps. To better understand this result, we conducted an additional analysis in which we verified whether the low number of tests is something peculiar of certain categories of apps. As such, we followed the categorization given by the Google Play Store and split the dataset into groups of apps based on their category. Afterwards, we computed the number of tests for the apps in each category, also assessing if the number of tests in one of them is statistically different from the others: to this aim, we computed the Mann-Whitney test [53], a non-parametric statistical analysis which can measure the extent to which the differences in terms of number of test suites available in different app categories are significant. We found no significance in each of the verified tests, meaning that the number of test suites does not depend on the category a mobile app belongs to. This finding suggests that there exist no particular groups of developers who would benefit from automated testing methods and/or techniques.

Table 2: Granularity and type of tests 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 **RQ**$_1$, we classified tests 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 [7, 12]. 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 [17, 38] and (2) performance testing is a more delicate problem than for traditional applications [44, 62], 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 [59, 67]. 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 [92]: the small amount of tests in this category suggests that more research would be needed in order to understand the reason behind these achievements [54]. 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 [23, 27, 76]: this is particularly worrying in the case of security, also considering the recent data provided by NOWSECURE[3], 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 [30, 49], 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 three main categories, namely, *GUI*, *Storage*, and *Other*: 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 the table, we can observe that most of the tests in the dataset refer to tests exercising the application logic of mobile apps. Behind this result, there might be different explanations: First, developers are not properly supported nor aware of currently available techniques when it comes to the testing of other aspects of their apps [16]. Second, it seems that they prefer to focus more on the testing of the main functionalities of their apps rather than storage or GUI-related classes. Finally, 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 [76].

> ***Finding 1***. *Mobile applications are poorly tested and, indeed, only 59% of the apps contain at least one test suite. As for the apps tested, most of the tests pertain to unit tests that exercise the functionalities of the app, while other aspects like, for instance, performance of GUI testing, are not widely considered.*

4 $\mathbf{RQ}_2$ - ON THE DESIGN OF TEST CASES IN MOBILE APPS

In this section we report methodology and results of our analysis aimed at addressing $\mathbf{RQ}_2$.

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 1,050 mobile apps. To assess the design of the considered test cases we covered two aspects that can characterize their maintainability and understandability. Table 4 summarizes the metrics adopted to address $\mathbf{RQ}_2$. Firstly, we computed test code quality metrics, relying on the metric suite originally defined by Chidamber and Kemerer [13] and other metrics related to code quality. We computed the Lines of Code (LOC): according to previous achievements [40, 74, 95], having higher size may cause issues for developers with respect to the maintainability of tests as well as to their fault-proneness [81, 87]. For similar reasons, we computed cohesion metrics such as Lack of Cohesion of Test Methods (LCOM5 [34]), Tight Class Cohesion (TCC), and Loose Class Cohesion (LCC) [15]; we measured different metrics as they can provide orthogonal information that may be useful to analyze the cohesion of tests better [86]. Furthermore, we considered the coupling between tests, which is one of the most critical problems when comprehending test code [94]. To this aim, we computed the Information Flow Coupling (IFC), a metric that captures the relations between tests in terms of information exchanged [86] and is among the best suited for assessing the quality of tests [22]. Finally, we considered the complexity of test code. In this case, the rationale comes from previous studies [18, 63, 97] which showed that complexity metrics may be related to the defectiveness of test code as well as may lower the overall understandability of the target of tests [94]. 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.

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 [88]. On the one hand, test smells make test code more change- and fault-prone [81] as well as harder to comprehend and maintain [5]. On the other hand, test smells have been shown to be one of the primary causes behind test instabilities [71, 72], thus making them extremely harmful for developers [20]. 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 definition are provided in Table 4. To detect them, we employed the code metrics-based tool developed by Bavota *et al.* [5], which has shown to have high accuracy, close to 86% of F-Measure [5, 71, 73] and has been validated several times in previous work [24, 66, 69, 81], thus making us confident of its suitability for our study.

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

Table 3: Types of production classes tested.

Activity		Intent		Fragments		Storage		Other	
Abs.	Rel.	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.
190	5%	9	1%	51	1%	108	3%	2,896	89%

Table 4: 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
Test smells	Assertion Roulette	A test containing several assertions with no explanation
	Eager Test	A test method testing more methods of the production target
	Indirect Testing	A test interacting with the target via another object
	Resource Optimism	A test that make optimistic assumptions on the existence of external resources
	Mystery Guest	A test that use external resources (*e.g.,* files or databases)

Table 5: 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

4.2 Analysis of the Results

Table 5 reports the distributions for all the quality metrics considered in our second research question.

Looking at the achieved results, 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 [94], this phenomenon may potentially lead to undesired issues like, for instance, potential flakiness due to a test ordering problem, which appears when the correct execution of a test depends on the execution of another one [47].

Finally, we found that the considered metrics do not suggest particular insights when considering cohesion aspects of tests. Indeed, for all the cohesion metrics, the values are spread over 0 and 1, with a greater frequency in the extremities. This could indicate that the cohesion of tests does not clearly differ between mobile apps and traditional software projects. More likely, cohesion could be related to other aspects that are not part of this study (e.g., developers' expertise).

Turning the attention to test smells, Table 6 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 [4, 5, 31] and claim that test smells have a high diffuseness also when considering the mobile context. Most of 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 [88]. Instances of *Eager Test* are also quite diffused and affect 31% of the test suites in our dataset. According to previous results [81], this smell type is associated with a lower effectiveness of the affected test in terms of fault detection capabilities. As for the other test smells, we found them to be 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.* [4] and Grano *et al.* [31], 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 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 [81]. 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 defined.

Table 6: Absolute and relative number of test smells detected.

Assertion Roulette		Eager Test		Mystery Guest		Resource Optimism		Indirect Testing	
Abs.	Rel.	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.	Abs.	Rel.
2,508	50%	1,556	31%	439	9%	123	3%	371	7%

Finding 2. *The metric profile of the considered test suites does not always indicate the presence of possible understandability and maintainability 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 last 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 statement coverage [93] and assertion density [43]. The former measures the amount of lines of production code touched by a test suite during its execution: we employed JaCoCo,[4] 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 Kudrjavets *et al.* [43]. We considered this metrics as the number of `assert` statements per test class KLOC has been associated in the past with a reduction of defect density in production code [11, 43], hence providing an indication of how good a test suite can actually be. To compute this metric, we developed our own tool, which we made available in our online appendix [8].

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. These values relate to low effectiveness [50] and, as a consequence, they can indicate that test suites developed in the context of mobile apps

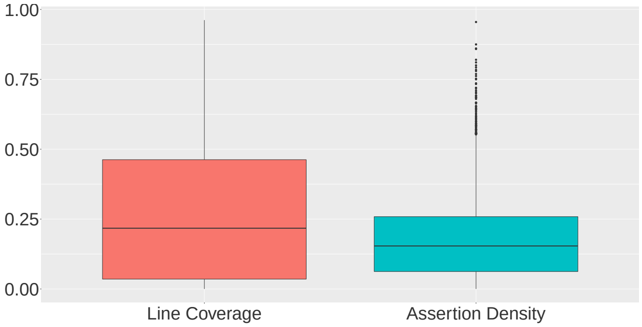

Figure 1: Distribution of test code quality metrics in our dataset.

have low capabilities of finding faults in production code. Our results complement previous findings in the field that showed how mobile apps tend to be more fault-prone than traditional applications and that most of the faults are introduced during the first stages of the development [83, 84]: likely, the limited amount of tests developed, along with their low effectiveness, represent two key reasons behind such a higher fault-proneness. Further empirical investigations into the relation between presence/effectiveness of tests and fault-proneness of production code would be worthy to better understand the impact of testing on source code quality and reliability—such an analysis is part of our future research agenda.

Nevertheless, we also observe a notable number of outliers, especially when considering the assertion density. One of them is represented by the test suite `tb.ProbeResultTest` contained in the DroidFish app: this test has 216 lines of code with 170 assertions (density=0.79). In the past history of the app, the corresponding production class, *i.e.*, `ProbeResult`, was frequently modified (*i.e.*, 298 commits, out of the total 875) without being affected by any fault. Of course, this result may be also due to other reasons, for instance the experience of the development team or the absence of design issues in the source code; however, the high assertion density of the test may have likely provided an effective guard against the introduction of defects.

On the contrary, the test suite named `jSscAdapterTest`, belonging to the package `tt.astronomy` of the Jtt app, has an assertion density close to zero (*i.e.*, 0.01) since it has just 1 assertion over 77 lines of code. Looking at the change history of the project, we observed that the corresponding production class, `SscAdapter`, has been the subject of 6 faults. All in all, our qualitative additional analyses somehow suggests that keeping test code effectiveness under control may substantially improve the quality of mobile applications and in cases where developers are keen to develop test cases, they have benefits in terms of fault-proneness of classes—confirming previous findings in the field [43, 57].

[4] https://www.jacoco.org

When it turns to code coverage, the discussion is similar. The vast majority of the test suites have low coverage and cannot properly exercise the corresponding production code. In this case, however, we noticed something different: 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 case of 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.* [45] 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 how mobile programmers still experience troubles when it comes to the development of effective tests.

***Finding 3**. Test suites are mostly not effective: the median code coverage and assertion density are 0.23 and 0.17, respectively. Our additional analyses revealed that, in cases where developers take care of their tests, they seem to be rewarded with a reduction of production code fault-proneness. Furthermore, we found that in some cases developers perceive code coverage as highly relevant to accept pull requests.*

6 DISCUSSION, IMPLICATIONS, AND LIMITATIONS

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

6.1 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 are poorly tested. The first, worrisome result of our study clearly indicated the lack of testing of mobile applications: not only the median number of tests is 2, but also the percentage of apps that do not contain any test is dangerously high (41%). 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 [91]. At the same time, the need for testing may be seen as secondary with respect to the development of production code, as shown by Beller *et al.* [6]. This is not only true in general, but it seems to affect mobile developers as well. For instance, let consider the case of Acastus Photon,[5] 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 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. As an example, 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: currently, there exist tools to automate GUI testing (Monkey or Sapienz [49]) while only a few automated and practical mechanisms are available for the generation of functional and non-functional test cases (*e.g.*, Evosuite [21]). As such, our findings further support the research in the field and outline, once again, the need for further approaches and tools than can automate testing and ease the developer's activities.

On integration and system testing. According to our findings, most of the test suites present in mobile apps pertain to unit testing, while only a limited amount of them refer to integration and system testing [2]. On the one hand, this result somehow confirms the difficulties that developers have when writing these types of test cases [2, 32, 75]. On the other hand, the lack of automated tools and/or support mechanisms likely influences this aspect: as such, our findings represent a call for novel approaches able to work at a higher-level with respect to existing ones that support developers while developing unit tests [21].

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. Given the high importance of these aspects in the today world, we argue that more methods to control for them should be developed. While the research community has been working already on them (*e.g.*, [19]), techniques that automate the detection of non-functional faults should be further investigated.

The test 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 [72, 81]. To identify and remove them, some test smell detectors have been developed in the past [33, 73, 90], however there are still two key limitations: (1) none of them has been actually tested in the context of mobile apps, so their accuracy is unknown; (2) according to previous findings [90], the existing detectors have limited detection capabilities. At the same time, techniques to refactor

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

test code are still unavailable. As such, the practical usage of test smell detectors and refactoring is precluded.

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, and complexity aspects. As an outcome, we found out a contradiction between the metric profile of tests and the actual presence of design issues. Indeed, 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. This may potentially indicate that researchers should go beyond currently available code metrics and define novel indicators that can better quantify the quality of test cases.

On the effectiveness of test suites. The last discussion point relates to the low effectiveness of the test cases analyzed, under all perspectives treated, *i.e.*, code coverage and assertion density. Our results clearly point out that, not only apps are poorly tested, but the available tests are also not effective and likely to miss faults in production code. On the one hand, our findings may be interesting for practitioners and raise their awareness of the status of mobile app testing. On the other hand, this is a further signal of the need for automated solutions that can support developers when performing testing activities.

6.2 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 [25, 26], 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 **RQ**$_1$ we employed the test-to-code traceability approach defined by Van Rompaey and Demeyer [89] 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 **RQ**$_2$, we employed an automatic test smell detector: its accuracy has been previously assessed [5, 71, 73] 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. 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, other than made available in our appendix [8].

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; given our goals and settings, the usage of statistics or of a mixed-method research approach would not have provided additional benefits. 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 statement coverage and assertion density, without considering another well-known indicator such as the mutation score [35], *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[6]) 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.* [28] reported statement 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.

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 [77] 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.

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.

7 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

[6] https://pitest.org

standard applications [91, 96], has pushed the research community to define methods to support developers with testing activities [60]. Furthermore, researchers have been investigating how developers test their mobile applications, also in comparison to standard systems [60], showing dissimilarities, peculiarity and possible effective practices. Given the goals of our paper, in this section we mainly focus on the studies that have tried to analyze the testing practices of mobile developers by (i) surveying and/or interviewing practitioners [39, 45] and (ii) performing mining software repository studies [16, 39].

Linares-Vásquez *et al.* [45] 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 [46] 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.* [14], which benchmarked automated test input generation tools, discovering that despite the research effort spent so far, 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.* [39] 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.* [36] 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.* [79] 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 [79]. Finally, a recent study by Cruz *et al.* [16] 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.

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: 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 [16, 39]: our dataset was indeed composed of over 1,700 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.* [39], 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 [11, 43].

8 CONCLUSION

In this paper, we investigated the characteristics of test suites written by developers of mobile applications under three 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, and (3) what is the effectiveness of tests, considering assertion density and code coverage. The main results of the study highlight that 59% 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. Finally, their effectiveness is low when considering all the computed metrics. Our findings provide a number of implications for researchers, in particular on the need for specific testing tools (*e.g.*, energy or performance approaches) and on the need for studies bringing evidence to developers with respect to the usefulness of testing for the success of mobile applications.

To sum up, our paper made the following contributions:

(1) A large-scale empirical study on the prominence, design quality, and effectiveness of test cases manually written by developers in the context of mobile applications;

(2) A research roadmap on the topic, which can be exploited by fellow researchers to delineate the future steps that may be performed to ease the testing activities of mobile developers;

(3) An online appendix [8] 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 follow the roadmap defined in Section 6.1 and includes the definition of 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).

ACKNOWLEDGMENTS

Hidden for double-blind review.

REFERENCES

[1] Jean-Yves Antoine, Jeanne Villaneau, and Anaïs Lefeuvre. 2014. Weighted Krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation.

[2] Gergő Balogh, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. 2016. Are my unit tests in the right package?. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 137–146.

[3] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering* 41, 4 (2014), 384–407.

[4] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. 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, 56–65.

[5] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.

[6] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering* 45, 3 (2017), 261–284.

[7] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, How, and Why Developers (Do Not) Test in Their IDEs *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/2786805.2786843

[8] Blinded. 2020. Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps — Online Appendix. https://figshare.com/s/0800d00b1c2d67e15915.

[9] Gemma Catolino. 2018. Does source code quality reflect the ratings of apps?. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 43–44.

[10] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. 2019. Cross-Project Just-in-Time Bug Prediction for Mobile Apps: An Empirical Assessment. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 99–110.

[11] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. 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, to appear.

[12] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.

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

[14] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering ASE*. IEEE, 429–440.

[15] Steve Counsell, Stephen Swift, and Jason Crampton. 2006. The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15, 2 (2006), 123–149.

[16] Luis Cruz, Rui Abreu, and David Lo. 2019. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering* (2019), 1–31.

[17] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. 2016. A quantitative and qualitative investigation of performance-related commits in android apps. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 443–447.

[18] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. 2018. A developer centered bug prediction model. *IEEE Transactions on Software Engineering* 44, 1 (2018), 5–24.

[19] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. 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, 103–114.

[20] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. (2019), to appear.

[21] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software *(ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[22] Enrico Fregnan, Tobias Baum, Fabio Palomba, and Alberto Bacchelli. 2018. A survey on software coupling relations and tools. *Information and Software Technology* (2018).

[23] Jerry Gao, Wei-Tek Tsai, Ray Paul, Xiaoying Bai, and Tadahiro Uehara. 2014. Mobile Testing-as-a-Service (MTaaS)–Infrastructures, Issues, Solutions and Needs. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE, 158–167.

[24] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software* 138 (2018), 52–81.

[25] Franz-Xaver Geiger and Ivano Malavolta. 2018. Datasets of Android Applications: a Literature Review. *arXiv preprint arXiv:1809.10069* (2018).

[26] Franz-Xaver Geiger, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli. 2018. A graph-based dataset of commit history of real-world android apps. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 30–33.

[27] Peter Gilbert, Byung-Gon Chun, Landon P Cox, and Jaeyeon Jung. 2011. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*. ACM, 21–26.

[28] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. 2017. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability* 66, 3 (2017), 854–874.

[29] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 72–82.

[30] Giovanni Grano, Adelina Ciurumelea, Sebastiano Panichella, Fabio Palomba, and Harald C Gall. 2018. Exploring the integration of user feedback in automated testing of android applications. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 72–83.

[31] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software* 156 (2019), 312–327.

[32] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2012. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 244–254.

[33] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. 2013. Automated detection of test fixture strategies and smells. In *Software Testing, Verification and Validation (ICST)*. 322–331.

[34] Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. 1996. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object oriented systems* 3, 3 (1996), 143–158.

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

[36] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real challenges in mobile app development. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 15–24.

[37] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. 2014. What do mobile app users complain about? *IEEE Software* 32, 3 (2014), 70–77.

[38] Heejin Kim, Byoungju Choi, and W Eric Wong. 2009. Performance testing of mobile applications at the unit test level. In *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*. IEEE, 171–180.

[39] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation ICST*. IEEE, 1–10.

[40] A Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. 2009. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering* 35, 2 (2009), 293–304.

[41] Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications.

[42] Daniel E Krutz, Mehdi Mirakhorli, Samuel A Malachowsky, Andres Ruiz, Jacob Peterson, Andrew Filipski, and Jared Smith. 2015. A dataset of open-source Android applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 522–525.

[43] Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. 2006. Assessing the relationship between software assertions and faults: An empirical investigation. In *2006 17th International Symposium on Software Reliability Engineering*. IEEE, 204–212.

[44] Christoph Laaber and Philipp Leitner. 2018. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 119–130.

[45] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution ICSME*. IEEE, 613–622.

[46] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. 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, 399–410.

[47] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 643–653.

[48] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.

[49] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.

[50] Brian Marick et al. 1999. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*. 16–18.

[51] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2016. A survey of app store analysis for software engineering. *IEEE transactions on software engineering* 43, 9 (2016), 817–847.

[52] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. 2016. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering* 21, 3 (2016), 1346–1370.

[53] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[54] Ali Mesbah and Mukul R Prasad. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 561–570.

[55] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[56] Roberto Minelli and Michele Lanza. 2013. Software Analytics for Mobile Applications–Insights & Lessons Learned. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 144–153.

[57] Audris Mockus, Nachiappan Nagappan, and Trung T Dinh-Trong. 2009. Test coverage and post-verification defects: A multiple case study. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 291–301.

[58] Israel J Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E Hassan. 2013. A large-scale empirical study on software reuse in mobile apps. *IEEE software* 31, 2 (2013), 78–86.

[59] Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. 2016. Anti-patterns and the energy efficiency of Android applications. *arXiv preprint arXiv:1610.05711* (2016).

[60] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE Press, 29–35.

[61] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[62] Meiyappan Nagappan and Emad Shihab. 2016. 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, 21–32.

[63] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. 2010. Change bursts as defect predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 309–318.

[64] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. 2016. 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, 552–562.

[65] Business of Apps. [n.d.]. *There are 12 million mobile developers worldwide, and nearly half develop for Android first.*

[66] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. 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, 5–14.

[67] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2019. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology* 105 (2019), 43–55.

[68] Fabio Palomba, Mario Linares-Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2018. Crowdsourcing user reviews to support the evolution of mobile apps. *Journal of Systems and Software* 137 (2018), 143–162.

[69] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: What if test code quality matters?. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 130–141.

[70] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. 2017. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th international conference on software engineering*. IEEE Press, 106–117.

[71] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 1–12.

[72] Fabio Palomba and Andy Zaidman. 2019. The smell of fear: On the relation between test smells and flaky tests. *Empirical Software Engineering* (2019), 1–40.

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

[74] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 176–185.

[75] Mauro Pezze, Konstantin Rubinov, and Jochen Wuttke. 2013. Generating effective integration test cases from unit ones. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 11–20.

[76] Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. 2014. 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, 30–40.

[77] Nick Pidgeon and Karen Henwood. 2004. *Grounded theory*. na.

[78] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Filomena Ferrucci. 2019. Third-party libraries in mobile apps. *Empirical Software Engineering* (2019), 1–37.

[79] Davi Bernardo Silva, Andre Takeshi Endo, Marcelo Medeiros Eler, and Vinicius HS Durelli. 2016. An analysis of automated tests for mobile Android applications. In *2016 XLII Latin American Computing Conference CLEI*. IEEE, 1–9.

[80] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. 2019. Test-driven code review: an empirical study. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 1061–1072.

[81] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 1–12.

[82] Statista. 2018. *Number of apps available in leading app stores as of October 2018.*

[83] Mark D Syer, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. 2015. Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal* 23, 3 (2015), 485–508.

[84] Mark D Syer, Meiyappan Nagappan, Ahmed E Hassan, and Bram Adams. 2013. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source Android apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 283–297.

[85] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 4–15.

[86] Bela Ujhazi, Rudolf Ferenc, Denys Poshyvanyk, and Tibor Gyimothy. 2010. New conceptual coupling and cohesion metrics for object-oriented systems. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, 33–42.

[87] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An empirical study of bugs in test code. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 101–110.

[88] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. 92–95.

[89] Bart Van Rompaey and Serge Demeyer. 2009. Establishing traceability links between unit test cases and units under test. In *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 209–218.

[90] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. 2007. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering* 33, 12 (2007), 800–817.

[91] Tony Wasserman. 2010. Software engineering issues for mobile application development. (2010).

[92] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 226–237.

[93] Yi Wei, Bertrand Meyer, and Manuel Oriol. 2012. Is branch coverage a good measure of testing effectiveness? In *Empirical Software Engineering and Verification*. Springer, 194–212.

[94] Chak Shun Yu, Christoph Treude, and Mauricio Aniche. 2019. Comprehending Test Code: An Empirical Study. (2019), to appear.

[95] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 17–23.

[96] Jack Zhang, Shikhar Sagar, and Emad Shihab. 2013. The evolution of mobile apps: An exploratory study. In *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile*. ACM, 1–8.

[97] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 9–9.