An Exploratory Study on the Refactoring of Unit Test Files in Android Applications

Anthony Peruma
axp6201@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Christian D. Newman
cnewman@se.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Mohamed Wiem Mkaouer
mwmvse@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Ali Ouni
ali.ouni@etsmtl.ca
ETS Montreal, University of Quebec
Montreal, Quebec, Canada

Fabio Palomba
fpalomba@unisa.it
SeSa Lab - University of Salerno
Fisciano (SA), Italy

ABSTRACT

An essential activity of software maintenance is the refactoring of source code. Refactoring operations enable developers to take necessary actions to correct bad programming practices (i.e., smells) in the source code of both production and test files. With unit testing being a vital and fundamental part of ensuring the quality of a system, developers must address smelly test code. In this paper, we empirically explore the impact and relationship between refactoring operations and test smells in 250 open-source Android applications (apps). Our experiments showed that the type of refactoring operations performed by developers on test files differ from those performed on non-test files. Further, results around test smells show a co-occurrence between certain smell types and refactorings, and how refactorings are utilized to eliminate smells. Findings from this study will not only further our knowledge of refactoring operations on test files, but will also help developers in understanding the possible ways on how to maintain their apps.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**.

KEYWORDS

Software maintenance and evolution, Unit testing, Test smells, Refactoring, Android applications.

ACM Reference Format:
Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. An Exploratory Study on the Refactoring of Unit Test Files in Android Applications. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3387940.3392189

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392189

1 INTRODUCTION

Modern software systems shield their production code with a quality gate, ensuring that proposed changes undergo several testing strategies before being shipped to end-users. Therefore, just like production code, any upgrade to software requirements would eventually trigger the evolution of test code, ensuring it matches the actual software behavior, and avoiding deprecation and decay [22]. Hence, test file designs must be of the highest quality in order to ease their maintenance and evolution [17, 26]. The main threat to test files design, just like production code, is the existence of test smells; symptoms of bad programming practices [35]. Test smells have been proven to threaten the quality of tests, making them harder to understand and to maintain by increasing their flakiness (i.e., tests that have a non-deterministic outcome) [25, 38, 39, 49]. Refactoring is one way to remove smells. In this context, refactoring is a disciplined software engineering practice to improve the internal design of software systems without altering its external behavior; it involves locating code smells and treating them as potential refactoring opportunities [23, 32].

Although the refactoring of production code has been the subject of several studies, we still notice a lack of investigations on whether and how developers typically refactor test code. Studying refactoring practices may be particularly interesting in the context of mobile applications (apps), which are the primary means for billions of users to communicate and interact with external services and other people [21]. There are over 2.5 million apps on the Google Play store (as of December 2019) [1, 34]. As a matter of fact, it is now, more than ever, essential for app developers to continually maintain their apps in order to reduce the risk of end-users switching over to a competing app [24, 37]. These apps, like traditional systems, are also susceptible to smells in their source code, and similarly, are impacted by the problems caused by bad smells [19]. At present, little knowledge is available on the impact of refactoring activities on test smells present in the test files of Android apps.

In this paper, we initiate the research on refactoring test suites, through performing an exploratory study on how developers refactor test code. In this work, we monitor the refactoring activities of test files, belonging to 250 open-source Android apps. Our mining procedure allows us to determine the most frequent refactoring operations typically applied to test code, as well as the most frequent refactoring operations applied to production code. Thus, allowing

us to compare and contrast the types of refactoring operations applied in these different contexts. Our study also identifies the type of refactoring operations that are eventually applied specifically to smelly test files, and whether their application is responsible for the removal of these smell instances.

1.1 Goal and Research Questions

The goal of this paper is to understand the relationship between refactoring changes and their effect on test smells. The results of this paper will serve to better understand how refactoring activities influence the stability of test smells (e.g., whether they increase or decrease in frequency) and ultimately recommend refactorings to remove (or at least avoid adding) additional test smells. Hence, we first explore the refactoring activities of Android app developers; more specifically, we target the test suite of apps. Therefore, our study aims at answering the following research questions:

RQ_1: What types of refactoring operations are applied to unit test files compared to non-test files? This research question looks into the type of refactoring operations applied by developers to test and non-test files. We want to know if developers treat test files differently from non-test files when it comes to refactoring activities. Knowing if there are differences in how developers refactor these two types of files will help us to better tailor our research methodology.

RQ_2: What types of refactoring operations are frequently applied to smelly test files? With this question, we aim to understand what types of refactorings tend to co-occur with a test smell. This research question will help us identify developer patterns between refactoring operations and test smells.

RQ_3: What kinds of refactorings are typically used to remove test smells? It is generally known that refactoring of source code is the mechanism to eliminate smells. From this question, we aim to understand the types of refactoring operations that are involved in the removal of test smells from the test suite. Findings from this research question will help us better focus our research on selecting refactoring operations that improve the maintainability of test suites.

1.2 Study Contributions

From this study, our main contributions to the field can be summarized as follows:

(1) An understanding and listing of refactoring operations applied to test suites of Android apps;
(2) Insights into the relationships that exist between refactoring operations and test smells in the test suites of Android apps;
(3) A comprehensive dataset for replication and extension purposes, available on our project website [2].

2 RELATED WORK

While there is an ample amount of existing studies around test smells, our related work is limited to studies that included aspects of test code refactoring in the study of test smells.

van Deursen et al. [53] introduced the concept of test smells as a result of studying refactoring of test code. The authors observed that refactoring of test code is different from production code due to test code having issues that are not usually found in production code and hence requiring special handling. In [52], van Deursen and Moonen proposed the concept of test first refactoring. This approach calls for the analysis of the test code in order to identify refactoring opportunities in the production code. Additionally, the authors also proposed a taxonomy of refactoring operations for test code. An approach to safely refactor test code was proposed by Guerra and Fernandes [29]. The authors utilize graphical notation to represent elements within a test suite along with a catalog of 15 test code specific refactorings to help developers understand test code refactorings and also perform refactorings more safely. Meszaros [35], provided developers with patterns and practices on how to write maintainable tests. The author also highlighted smells that are exclusive to test code along with instructions on how to refactor such test cases.

In [38], Palomba and Zaidman investigated the impact of refactoring on flaky tests. Flaky tests are tests that can exhibit failing and passing results with the same code. Through their study, the authors observed that flaky tests are prevalent in systems with around 45% of test methods exhibiting a form of flakiness. The authors also report that refactoring actions performed on the test code help in resolving a majority of flaky tests. In addition to developing a test smell detection tool, TestHound, Greiler et al. [27] also looked into the types of refactoring that can be applied to correct the smelly test code. Focusing on test fixture based smells, Greiler et al. [28] investigated the evolution of the smell throughout the lifetime of the project along with providing strategies and recommendations for avoiding and correcting such smells. To better support developers in understanding the structure of their test suite with relation to test smells, Breugelmans and Van Rompaey implemented a tool called TestQ [18]. By providing a visualization of the test suite, developers will be able to detect refactoring opportunities easily and faster.

Bavota et al. [17] studied the impact of test smells with regards to code comprehension. In their study, the authors refactored smelly test code to remove the test smell. The authors report that participants in their study reported a negative code comprehension experience when performing maintenance activities on the smelly code. A developer survey conducted by Tufano et al. [51]. It was interesting to note that the participants of the survey did not perceive test smells as problematic. Additionally, the participants were of the view that refactoring of the smelly test code would not be advantageous to the design of the test suite. A study on developer reactions to test smells was conducted by Schvarcbacher et al. [47]. In this study, the authors integrate the same test smell detection tool used in our study into a code quality monitoring system. Preliminary results from their study showed that while developers agree that the smells are problematic, they prefer to refactor their test suites only to correct a select set of the smell types.

Finally, as our study focuses on Android apps, we looked into studies that investigated the refactoring of such systems. We observed that the majority of the studies do not focus on granular source code refactoring operations. Instead, the work mostly revolves around API level refactoring. In other words, the studies looked at mechanisms to improve the quality characteristics of the apps (such as energy efficiency, concurrency, etc.) by recommending alternate API's or design patterns ([16, 20, 30, 33, 34, 55]).

Table 1: Summary of the different test smell types exhibited by files contained in the dataset [42] used in our study.

Test Smell	Detection Rule
Assertion Roulette	A test method contains more than one assertion statement without an explanation/message (parameter in the assertion method)
Conditional Test Logic	A test method that contains one or more control statements (i.e., if, switch, conditional expression, for, foreach and while statement)
Constructor Initialization	A test class that contains a constructor declaration
Default Test	A test class is named either `'ExampleUnitTest'` or `'ExampleInstrumentedTest'`
Duplicate Assert	A test method that contains more than one assertion statement with the same parameters
Eager Test	A test method contains multiple calls to multiple production methods
Empty Test	A test method that does not contain a single executable statement
Exception Handling	A test method that contains either a throw statement or a catch clause
General Fixture	Not all fields instantiated within the `setUp` method of a test class are utilized by all test methods in the same test class
Ignored Test	A test method or class that contains the `@Ignore` annotation
Lazy Test	Multiple test methods calling the same production method
Magic Number Test	An assertion method that contains a numeric literal as an argument
Mystery Guest	A test method containing object instances of files and databases classes
Redundant Print	A test method that invokes either the `print` or `println` or `printf` or `write` method of the `System` class
Redundant Assertion	A test method that contains an assertion statement in which the expected and actual parameters are the same
Resource Optimism	A test method utilizes an instance of a `File` class without calling the method `exists()`, `isFile()` or `notExists()` methods of the object
Sensitive Equality	A test method invokes the `toString()` method of an object
Sleepy Test	A test method that invokes the `Thread.sleep()` method
Unknown Test	A test method that does not contain a single assertion statement and `@Test(expected)` annotation parameter

From a refactoring operation perspective, Park et al. [40] applied refactoring operations on source code to investigate if the operations help with the energy consumption of mobile devices. The authors observed that only a partial set of refactoring operations help conserve energy; these findings were later extended by Sahin et al. [46] in the context of an empirical study on the impact of the Fowler's refactoring actions [23]. Later, Palomba et al. [36] focused on refactoring operations specifically designed to remove Android-specific code smells. They reported that (*i*) methods of mobile apps affected by energy-specific code smells consume notably more than methods not affected by any design problem and (*ii*) the refactoring operations associated to these smells can increase the energy efficiency of mobile applications in most cases.

In a preliminary study of refactoring operations occurring in Android apps, Peruma [41], observed that rename operations are the most common type of refactoring applied to source code. Furthermore, by studying the commit log messages, the author observed that app developers perform refactorings to improve code readability, fix defects, and enhance system design.

3 METHODOLOGY

The general methodology of our study is depicted in Figure 1, and consists of utilizing a mining tool for the detection of refactoring operations from an existing dataset of unit test files and smells in Android apps.

We utilized the dataset of Android test smells generated from our previous study [42] for the experiments in this study. The dataset contains details of 19 different types of test smells that were prevalent in the Junit-based unit test files of 656 open-source Android apps. The dataset captures the types of test smells that occur throughout the history of each unit test file throughout the lifetime of the project. In addition to test files, the dataset also contains the production file associated with the test file (where applicable). In total, 206,598 JUnit based unit test files were processed and resulted

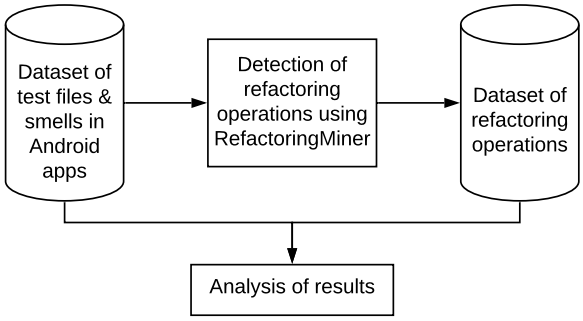

Figure 1: Overview of our methodology.

in the analysis of 1,187,055 unit test methods. Table 1 provides a summary of the types of test smells that are part of the dataset. Please refer to [42] for a more detailed definition of these smells.

We utilized the tool, RefactoringMiner [50], to detect the different refactoring operations applied by developers to the source code. To detect refactoring operations, RefactoringMiner enumerates over the entire commit history of a project and compares the changes made to the source code. Using a pre-defined set of refactoring rules, RefactoringMiner checks if the changes made to the source file can be categorized as a refactoring. The version of RefactoringMiner utilized for this study can detect 39 types of refactoring operations. We executed RefactoringMiner on the set of apps contained in our dataset of test smells. In total, we detected 614 apps that underwent refactorings. After the removal of outliers (via the Tukey Fences approach [31]), each app, exhibited on average 321.4 refactoring operations. A statistical summary of these results is provided in Table 2. It is important to highlight that the selection of RefactoringMiner was based on two observations: on the one hand, it represents state

Table 2: Statistical summary of refactoring operations associated with each app in the dataset.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1	33	139	321.4	466	1673

Table 3: Statistical summary of refactoring operations associated with each app for test and non-test files.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Refactorings applied to test files					
1	2	7	13.99	19	71
Refactorings applied to non-test files					
1	146.5	439.5	743.8	1161.2	3278.0

of the art in the field of refactoring detection [54]; on the other hand, its empirical validation [50] showed that the tool is able to reconstruct refactoring operations with an F-Measure close to 93%, hence being particularly suitable in the context of a large-scale mining investigations like the ours.

4 EXPERIMENTAL RESULTS

From the 614 apps that underwent refactorings, only 250 apps contained test files that underwent a refactoring. In total, we observed that only 4,709 of test files that were part of the 250 apps were refactored by developers. For the 250 apps that do contain test files, we looked at the distribution of refactorings occurring in these test files. As shown in Table 3, after the removal of outliers (via the Tukey Fences approach [31]), on average, each app had 13.99 refactoring operations performed on test files. However, the average for the same set of apps based on the refactorings applied to non-test files is much larger. Going forward, our experiments will utilize these 250 apps to answer our research questions.

4.1 RQ_1: What types of refactoring operations are applied to unit test files compared to non-test files?

In total, we detected 34 different types of refactoring operations that developers apply to test files. The most common type is *Rename Method*, which occurs approximately 19.81% over the total number of applied refactorings in the dataset. Due to space constraints, we present the top five occurring refactoring operations in Table 4. However, it is interesting to note that these five operations account for almost 65% of the operations applied by developers. Furthermore, the majority of the refactoring operations are clustered around data type changes and identifier renames. Approximately 38.84% of operations are renames, while type changes account for 34.87%; the remaining operations account for 26.29%. The high occurrence of renames is not surprising, as prior research has shown that renames are one of the most common types of refactorings applied by developers to source code [43–45, 54]. However, other than for API migration-related studies, at present, research on type change refactorings are not yet available.

Table 4: Refactoring operations applied to test files.

Refactoring Operation	Count	Percentage
Rename Method	1,511	19.81%
Change Variable Type	1,452	19.03%
Rename Variable	803	10.53%
Change Attribute Type	773	10.13%
Extract Method	426	5.58%
Other Operations	2,664	34.29%
Total	7,629	100%

For our comparison with non-test files, we looked at the remaining set of source files in the dataset that underwent a refactoring. All 39 refactoring operations that were supported by the version of RefactoringMiner used in this study were detected in this set of source code files. Interestingly, we observed that *Move Class* was the most popular refactoring operation for Android developers, occurring 13.33% of the time. However, unlike the test files, the top five refactorings only accounted for 42.35% of the refactorings. Again, looking at the clustering of rename and type change refactorings, we observed that 28.94% of the refactorings were clustered under renames, while 27.93% of refactorings were changes to the type of the identifier. The remaining refactorings accounted for 43.13% of the operations performed by developers. Additionally, we observed that four out of the top five test file based refactoring operations are related to methods or identifiers contained within methods. For non-test files, only three of the top five refactorings are related to methods, while the remaining are at the class-level.

Interestingly, for test files, if we exclusively look at refactorings applied to methods and classes, we notice that approximately 8.02% of these refactorings are applied to classes while the remaining 91.98% of operations are applied to methods. When it comes to non-test files, the distribution is different - method refactorings account to 49.27%, with 50.73% of the operations related to classes.

Finally, the entire list of refactoring operations applied to test and non-test files are available on our project website [2].

> **Summary.** We have shown that the types of refactorings applied to test and non-test files in Android apps are different. Non-test files are subject to more design level types of refactorings (e.g., *Move Class*), while test files tend to undergo more renames (e.g., *Rename Method*, *Rename Variable*) and data type changes (e.g., *Change Variable Type*) to identifiers. Furthermore, in test files, developers apply more refactorings to methods or other identifiers part of/contained within methods than to classes or attributes.

4.2 RQ_2: What types of refactoring operations are frequently applied to smelly test files?

To answer this research question, we extracted the list of smelly test files from the original dataset and then compared the results to the dataset of refactoring operations. We observed that 4,589 test files that had one or more smells had undergone a refactoring. To understand the types of refactorings that tend to co-occur with

Table 5: Refactoring operations applied to non-test files.

Refactoring Operation	Count	Percentage
Move Class	23,180	13.33%
Change Parameter Type	14,178	8.15%
Change Attribute Type	12,921	7.43%
Rename Method	12,074	6.94%
Rename Parameter	11,299	6.50%
Other Operations	100,249	57.65%
Total	173,901	100%

a smelly test file, we first extracted test files that exhibited only one type of smell and then looked at the refactorings applied to the file (if any). Due to space constraints, we present only a few of the results; the complete listing is available on our project website. Table 6 shows the frequency of co-occurrence of a subset of test smells and refactoring operations.

Looking at test files that exhibit the *Assertion Roulette* smell, we see the majority of refactoring operations performed by developers are related to a change in the data type of a variable within a method (i.e., *Change Variable Type*). Interestingly, the *Extract Method* refactoring frequently co-occurs with *Lazy Test* and *Eager Test* smells. Both these smells are due to developers violating testing best practices when calling methods in the production class/file. *Eager Test* is due to developers exercising multiple production methods in a single test method, while *Lazy Test* is when multiple test methods invoke the same production method. Most likely, the developer performs extensive design-related changes to the production code, which results in similar changes to the test suite.

Another interesting finding is the co-occurrence of the smell *General Fixture* with the refactoring operation *Change Attribute Type*. This particular smell arises when the test case fixture is too general, and the test methods only access part of it. Hence a developer, performing a type change to an attribute in the same test suite, should be mindful that there is a possibility of the attribute contributing to the emission of a *General Fixture* smell, if the same attribute is utilized in the `setUp()` method and not utilized by all test methods in the file. Looking at the smell *Redundant Assertion*, we see that *Move Method* is a frequent refactoring operation that co-occurs with this smell. *Redundant Assertion* is caused when test methods contain assertion statements that are either always true or false. This smell is mostly introduced due to mistakes made by the developer. A *Move Method* can be thought of as a design-level related refactoring and would most likely be more complicated to perform given the decisions a developer would need to make. Hence, there is a likely chance of developers forgetting to remove debugging related code such as those that result in a *Redundant Assertion* smell when implementing and verifying the *Move Method* refactoring.

Summary. We observed that there exist certain test smells and refactoring operations that co-occur frequently. For instance, the smells *Lazy Test* and *Eager Test* frequently co-occur with an *Extract Method* refactoring operation.

Table 6: Frequently co-occurring test smells and refactorings.

Smell Type	Co-occurring Refactoring Operation	Count	Percentage
Assertion Roulette	Change Variable Type	141 (Total: 266)	53.01%
Eager Test	Extract Method	14 (Total: 33)	42.42%
Lazy Test	Extract Method	20 (Total: 66)	30.30%
General Fixture	Change Attribute Type	8 (Total: 21)	38.10%
Redundant Assertion	Move Method	9 (Total: 31)	29.03%

4.3 RQ_3: What kinds of refactorings are typically used to remove test smells?

To answer this question, we looked at the lifetime history for each smelly test file. For each commit instance of such a file, we compared if the total smells exhibited by the file reduced and the amount by which it reduced. Next, we looked at the refactoring operations applied by the developer on the instance of the file that showed a reduction in smell count. From the dataset, we observed that 481 smelly test files that showed a reduction in smells also underwent a refactoring. On average, we observed that smells exhibited in a test file tend to reduce by 1.30 smells when the file is refactored. These same files, on average, exhibited around 2.98 test smells. Next, we looked at the number of refactoring operations applied to these test files. On average, 2.12 refactoring operations are applied to a smelly test file that results in a reduction in the smell count of the same file. Table 7 shows a statistical summary.

We also looked at the different combinations of smell reductions and refactoring operations applied to a smelly file. We observed that that the most frequent combination is the application of a single refactoring operation that would cause a single reduction of a smell type. This particular combination occurs 38.46% of the time. The next highest combination, at 12.89%, is the reduction of one smell type through the application of two refactoring operations.

For the single smell reduction, single refactoring combination, as shown in Table 8, we observed that the refactoring operation *Change Variable Type* is frequently involved in commits that also show a reduction in smell counts. Furthermore, from Table 9, the smell *Eager Test* is frequently resolved by developers when performing a single refactoring operation.

Due to space constraints, we report on a select set of observations of the commits that were involved in a single smell reduction, single refactoring combination. With regards to the *Eager Test* smell, when we look at commit [3], we observed that the method `testHumanDistance()` was exhibiting an *Eager Test* smell as this method was accessing a production method more than once. Due to a design-related change (i.e., "move...code to own class"), the developer eliminates this smell from the test file by performing a *Move Method* on the smelly test method. In another instance [4], the developer corrects an *Eager Test* smell by performing a *Change*

Variable Type. This action was performed by the developer to "fix unit test" in which a method associated with the new variable type, in the test method, is utilized instead of a method associated with the production object.

Looking at the *Assertion Roulette* smell, in commit [5], the developer performs a *Change Variable Type* on an object that was being used in an assertion method that lacked an explanation message. This refactoring action, performed by the developer to address "...lint recommendations", resulted in the developer removing the smelly assertion method from the test method. We also observed a *Rename Method* also utilized to resolve an *Assertion Roulette* smell as part of a bug fix. In commit [6], we observed that the developer renames test methods to be more specific in their behavior (e.g., `shouldSanitizeInputData` to `shouldSanitizeInputDataEmail`) and, as a result, removes assertion statements that do not reflect the behavior associated with the name of the test method.

We also noticed that the occurrence of *Extract Method* with the removal of the smell *Conditional Test Logic* is popular in our dataset. In [7, 8], we see developers extracting code containing a loop out of a test method. Looking at the messages associated with these two commits, we are made to understand that the code changes were made as part of either a change to or addition of functionality. In other words, the developers to do not explicitly state that their goal is to resolve the smell (or improve code comprehension).

The use of *Rename Method* helps alleviate the smell *Unknown Test*. In [9], the developer renames the test method to reflect its purpose better (`testGetCustomFields` to textttttestGetNullCustomFields) and includes an assertion statement to this effect. In [10], the developer has a test method, `testComplete`, without an assertion statement. By default, JUnit considers methods that start with the term ''test' as a test method and would automatically execute the method when the test suite is run. However, looking at the commit message, we see that the developer never intended this method to be a test method and after realizing that, "the name 'testComplete' will cause test runner to actually execute it as a test case, which is not the intention" renamed the method to `notifyComplete`.

We have also seen the use of *Change Attribute Type* involved in the indirect removal of certain smell types. For example in commit [11], by applying a *Change Attribute Type* operation, the developer eliminates the *Magic Number Test* smell from a test method. However, looking at the code, it seems that this co-occurrence is by chance; the replacement of magic numbers with constants is most likely due to adhering to coding standards to improve code comprehension. A similar pattern is seen in commit [12]. In this example, the smell *Ignored Test* is removed in the same commit in which the developer applies the same refactoring operation. However, a review of the code and commit message shows that resolving of this code is not related to refactoring operation. In another example, commit [13] shows a co-occurrence with the elimination of the smell *Redundant Print* and the refactoring operation *Replace Variable With Attribute*. However, looking at the code and commit message, we see that while the smell is indeed removed, it is debatable if the refactoring operation was solely responsible for the removal. As the main purpose of the commit was to fix a defective test, it is not possible to determine if the developer determined that the print statement in the test method was redundant and needed to be removed or if the refactoring operation resulted in the developer being forced to remove the statement to avoid a compilation or runtime error.

Table 7: Statistical summary of smell reductions and refactoring operations associated with each refactored smelly file.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Amount of smells reduced in smelly file					
1	1	1	1.30	2	3
Amount of refactoring operations applied to smelly file					
1	1	1	2.12	3	8

Looking at commit [14], we see that even though *Rename Variable* co-occurs with the removal of the smell *Constructor Initialization*, the refactoring operation does not contribute to the removal of the smell. From the code and commit message, we see that the refactoring operation is more to do with adhering to coding standards while the smell removal is due to the test being "Upgrade(d) test to JUnit 4." In commit [15], we observed that the smell *General Fixture* is removed when a *Move Attribute* operation is applied to the file. On observation of the code, we see that the refactoring operation did indeed resolve the smell as the attribute that was moved was being utilized within the test fixture method, but not all test methods were using it; hence, the existence of the smell.

From this RQ, we have seen that there exist scenarios where a test smell is eliminated when a refactoring operation is applied to a test file. On the surface, one might assume that developers perform refactoring operations to fix smelly code. However, on analysis of the code, this is not entirely true. As an exploratory study, our findings in this research question shows that more research is needed in this area. We have shown that refactorings do play a part in correcting smelly code in test files; however, more in-depth, and developer supported, studies are needed to fully understand how refactorings can be exclusively utilized to fix smelly code.

Summary. Despite their low frequency, there exist scenarios where refactoring operations are utilized to correct a test smell. However, these refactorings are applied by developers for reasons other than for the correction of smells; the fixing of smells is merely a byproduct. At most, a single smell is corrected by the application of a single refactoring with *Change Variable Type* being one of the most common refactorings applied when a smell is removed.

5 DISCUSSION & FUTURE DIRECTION

RQ$_1$ showed us that developers treat test and non-test files differently with regards to refactoring operations. For instance, non-test files undergo more design/structure level types of refactorings, unlike test files that are limited to somewhat cosmetic updates such as renames and type changes. Such findings should not be entirely surprising as test cases (i.e., test methods) are meant to exercise a minimal unit of source code and, therefore, would mostly require changes to data types and identifier names to match the new design level changes implemented in the non-test files. The high volume

Table 8: Top five refactoring operations that co-occur with the reduction of a single smell.

Refactoring Operation	Count	Percentage
Change Variable Type	65	35.14%
Rename Method	34	18.38%
Change Attribute Type	14	7.57%
Extract Method	14	7.57%
Move Method	13	7.03%
Other Operations	45	24.32%
Total	185	100%

Table 9: Top five smell types that are eliminated when a single refactoring is applied to a commit.

Smell Type	Count	Percentage
Eager Test	71	38.38%
General Fixture	16	8.65%
Lazy Test	13	7.03%
Magic Number Test	13	7.03%
Conditional Test Logic	12	6.49%
Other Operations	60	32.43%
Total	185	100%

of refactorings being applied to methods within test files should not be surprising as methods are the key components in a test suite. From these findings, developers would now be better prepared to understand the level of rework that would be involved when it comes to refactoring (non-)test files and hence aide in better project estimation planning. Future work in this direction can lead to more specialized refactoring tools for test and non-test source code.

In **RQ**$_2$, we explored the co-occurrence of refactorings and smells. Our findings showed interesting patterns, such as a high co-occurrence between method based refactorings and test smells. Based on the phenomenon that most refactorings are applied to test methods, developers should focus on test smells being exhibited by the methods they are refactoring. From our findings, we envision that app developers are now better prepared when refactoring test files with regards to smells. Developers will be aware of the most likely smell that exists or being introduced into the test suite when performing a refactoring. Taking this step further, researchers and/or tool vendors can be better equipped to offer an automated resolving of smells when a developer applies a specific refactoring to a test.

While a previous study [42] has shown that the volume of smells exhibited by a test file tends to remain steady over the lifetime of the file, from **RQ**$_3$—though low in volume—we observed situations where developers resolve smells exhibited by test files. Even though refactoring is supposed to remove smells, not all developers refactor their code with the aim of smell removal; the removal is mostly as a result of a byproduct of refactoring the test suite. Research by Tufano et al. [51] and Peruma et al. [42] showed that not all developers perceive all test smells as problematic. Hence, the refactoring operations performed on test files are mostly related to other types of development activities such as fixing issues, adhering to coding standards, and incorporating new/updated functionality. Even though we did notice patterns such as *Extract Method* resolving the *Conditional Test Logic* smell, more research is needed into this area, especially with developer involvement, to identify false-positive patterns and explore the developer's intention on proactively fixing smells with refactoring operations.

6 THREATS TO VALIDITY

A critical threat to this study is the domain for this study. This study is singularly focused on Android apps and may not be representative of non-mobile systems. However, given the volume of apps and ease of app development, it is important for app developers to be aware of the type of refactoring that they will be undertaking on their app during maintenance activities. Furthermore, as the test files are JUnit based, non-mobile developers (and researchers) can use our findings as a starting point when maintaining the test suite of non-mobile systems.

The original dataset can also be considered as a threat. However, the dataset has been used in a prior, published study on Android test suites [42]. Furthermore, at present, there does not exist any other datasets that focus on the test suite of Android apps. Additionally, the smell detection tool used to construct the dataset has also been utilized in another study on test smells [47]. The correctness of RefactoringMiner in the identification of refactoring operations is also a threat to this study. However, prior studies [48, 50] have shown that RefactoringMiner has high precision and recall scores when compared to similar open-source tools.

Finally, it is worth remarking that the conclusions drawn in our study are based on statistical data and co-occurrence analysis. As such, our study should be considered as a preliminary investigation into the way mobile developers perform refactoring operations. Hence, our findings should be complemented with further insights in order to shed light on the rationale pushing developers to perform such operations on test files, or when tests are affected by design issues. This kind of analysis is already part of our future research.

7 CONCLUSION & FUTURE WORK

In this study, we explored the refactoring operations applied by Android app developers on their apps. We utilized an existing dataset of open-source app test files, and test smells for our study. Additionally, we used RefactoringMiner to identify the refactoring operations in the dataset. Our analysis of 250 projects showed us that app developers apply a different set of refactorings to test and non-test source code files. Furthermore, we presented the common refactorings that co-occur with test smells, and finally, we looked at the smells that are resolved when a test file is refactored.

Going forward, our future work in this area will involve a more deep-dive into a select set of refactoring operations (i.e., operations that were frequently applied by developers) applied to test files. Additionally, we will also be looking at the type (i.e., experience) of developers that apply these types of refactorings and the specific reasons pushing them to apply certain refactoring operations.

REFERENCES

[1] [n.d.]. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[2] [n.d.]. https://testsmells.github.io.

[3] [n.d.]. https://github.com/cgeo/cgeo/commit/61b3c77#diff-d2ce8bcdab02bf8d70b96dc0e8956c1b.

[4] [n.d.]. https://github.com/open-keychain/open-keychain/commit/5d6c2d9#diff-17c72e4451fe562348f9c2a55fd0b257.

[5] [n.d.]. https://github.com/tilal6991/HoloIRC/commit/7b9405f#diff-37574423bd41b688fe5ef4f3ecaacc44.

[6] [n.d.]. https://github.com/jberkel/sms-backup-plus/commit/7884ec6#diff-22a08f2f0c231024bf25a4da696a40ef.

[7] [n.d.]. https://github.com/rtyley/agit/commit/6fd4b1c#diff-fb8ec323f48634266fb37e91894eb5c7.

[8] [n.d.]. https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer/commit/84ded72#diff-8380d0cb5481ea67fa154bd13572ebe3.

[9] [n.d.]. https://github.com/wordpress-mobile/WordPress-Android/commit/9c9691a#diff-ec439e6e6d7cb33ad174cd3e5b6da9bd.

[10] [n.d.]. https://github.com/CyanogenMod/android_packages_apps_Browser/commit/0622f96#diff-5feae667a2375978cec3bc0cfc9efd3b.

[11] [n.d.]. https://github.com/aragaer/jtt_android/commit/4a7e298#diff-6193986e47a81e5e53e7f13906c54095.

[12] [n.d.]. https://github.com/google/iosched/commit/26f90d3#diff-a860a9085797f69ceab90031a94ff63f.

[13] [n.d.]. https://github.com/k3b/APhotoManager/commit/6bcaf34#diff-21b786b93511440d8bfc28d560b58b88.

[14] [n.d.]. https://github.com/wikimedia/apps-android-wikipedia/commit/7895e4b#diff-fc2ee68efe30e2b8ae06036781654774.

[15] [n.d.]. https://github.com/andstatus/andstatus/commit/da49061#diff-d7a31ea4c55bae472368bccf882e4e0a.

[16] Abhijeet Banerjee and Abhik Roychoudhury. 2016. Automated Re-Factoring of Android Apps to Enhance Energy-Efficiency. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* (Austin, Texas) *(MOBILESoft '16)*. Association for Computing Machinery, New York, NY, USA.

[17] 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.

[18] Manuel Breugelmans and Bart Van Rompaey. 2008. TestQ: Exploring structural and maintenance characteristics of unit test suites. In *IN WASDETT-1*.

[19] Suelen Goularte Carvalho, Maurício Aniche, Júlio Veríssimo, Rafael S Durelli, and Marco Aurélio Gerosa. 2019. An empirical catalog of code smells for the presentation layer of Android apps. *Empirical Software Engineering* 24, 6 (2019).

[20] Luis Cruz, Rui Abreu, and Jean-Noël Rouvignac. 2017. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems* (Buenos Aires, Argentina) *(MOBILESoft '17)*. IEEE Press, 205–206.

[21] 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.

[22] Barrett Ens, Daniel Rea, Roiy Shpaner, Hadi Hemmati, James E Young, and Pourang Irani. 2014. Chronotwigger: A visual analytics tool for understanding source and test co-evolution. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 117–126.

[23] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[24] 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.

[25] 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).

[26] Giovanni Grano, Fabio Palomba, and Harald C Gall. 2019. Lightweight assessment of test-case effectiveness using source-code-quality indicators. *IEEE Transactions on Software Engineering* (2019).

[27] M. Greiler, A. van Deursen, and M. Storey. 2013. Automated Detection of Test Fixture Strategies and Smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 322–331.

[28] M. Greiler, A. Zaidman, A. van Deursen, and M. Storey. 2013. Strategies for avoiding text fixture smells during software evolution. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 387–396.

[29] E. M. Guerra and C. T. Fernandes. 2007. Refactoring Test Code Safely. In *International Conference on Software Engineering Advances (ICSEA 2007)*. 44–44.

[30] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. 2015. Detecting Antipatterns in Android Apps. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*. 148–149. https://doi.org/10.1109/MobileSoft.2015.38

[31] A.R. Jones. 2018. *Probability, Statistics and Other Frightening Stuff*. Taylor & Francis.

[32] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery.

[33] Y. Lin, S. Okur, and D. Dig. 2015. Study and Refactoring of Android Asynchronous Programming (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 224–235. https://doi.org/10.1109/ASE.2015.50

[34] Ivano Malavolta, Roberto Verdecchia, Bojan Filipovic, Magiel Bruntink, and Patricia Lago. 2018. How Maintainability Issues of Android Apps Evolve. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

[35] G. Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*. Pearson Education.

[36] 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.

[37] 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.

[38] F. Palomba and A. Zaidman. 2017. Does Refactoring of Test Smells Induce Fixing Flaky Tests?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–12. https://doi.org/10.1109/ICSME.2017.12

[39] Fabio Palomba and Andy Zaidman. 2019. The smell of fear: on the relation between test smells and flaky tests. *Empirical Software Engineering* (Oct 2019).

[40] Jae Jin Park, Jang-Eui Hong, and Sang-Ho Lee. 2014. Investigation for Software Power Consumption of Code Refactoring Techniques.. In *SEKE*. 717–722.

[41] A. Peruma. 2019. A Preliminary Study of Android Refactorings. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 148–149. https://doi.org/10.1109/MOBILESoft.2019.00030

[42] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., USA, 193–202.

[43] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. [n.d.]. Contextualizing Rename Decisions using Refactorings, Commit Messages, and Data Types. *Journal of Systems and Software* ([n. d.]).

[44] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. 2018. An Empirical Investigation of How and Why Developers Rename Identifiers. In *Proceedings of the 2nd International Workshop on Refactoring* (Montpellier, France) *(IWoR 2018)*. Association for Computing Machinery, New York, NY, USA, 26–33. https://doi.org/10.1145/3242163.3242169

[45] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. 2019. Contextualizing Rename Decisions using Refactorings and Commit Messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 74–85. https://doi.org/10.1109/SCAM.2019.00017

[46] Cagri Sahin, Lori Pollock, and James Clause. 2014. How do code refactorings affect energy usage?. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[47] Martin Schvarcbacher, Davide Spadini, Magiel Bruntink, and Ana Oprescu. 2019. Investigating developer perception on test smells using better code hub-Work in progress. In *2019 Seminar Series on Advanced Techniques and Tools for Software Evolution, SATTOSE 2019*.

[48] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery.

[49] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–12.

[50] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, 12.

[51] 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* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA.

[52] Arie Van Deursen and Leon Moonen. 2002. The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*. Citeseer, 71–76.

[53] 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 (XP)*. 92–95.

[54] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180 (2019).

[55] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. 2012. Refactoring Android Java Code for On-Demand Computation Offloading. *SIGPLAN Not.* 47, 10 (Oct. 2012), 233–248.