An Empirical Study on the Impact of Refactoring on Test Cases

Bill Chen
bill1119@hotmail.com
Concordia University

Steven Locke
swg.locke@gmail.com
Concordia University

Sophia Quach
sophiaquach16@gmail.com
Concordia University

ABSTRACT

Our empirical study examines existing GitHub projects and how the tests of these projects are affected by refactoring commits. Refactoring is known to have positive benefits to software quality, however it is possible that a refactoring commit can contain other changes such as introducing new functionality. Currently, little research has be conducted to determine the effects of refactoring on test cases. Due to this, it is important not to mistakenly attribute test failures to refactoring when actual causes are additional changes with the refactoring. Our approach examines refactorings done between two consecutive releases, analyzing the interesting refactoring commits and studying their impact on existing unit tests. For tangled refactorings, we create a further step by manually creating a new commit and isolating the core refactorings. We also define four types of manual testing, based on the nature of the refactoring and its test results. Our main findings are that purely clean refactorings did not cause any failures and for some of the commits analyzed, not only do they include a main refactoring, but also include new logic around them such as changes around the call sites. We also include a ruleset established on criteria that determines a distinction between a clean and dirty refactoring.

KEYWORDS

refactoring, software engineering, testing, empirical study

ACM Reference Format:
Bill Chen, Steven Locke, and Sophia Quach. 2020. An Empirical Study on the Impact of Refactoring on Test Cases. In *Proceedings of Fall 2019 (SOEN6491, Dr Tsantalis).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Our empirical study examines existing GitHub projects and how the tests of these projects are affected by refactoring commits. We will be focusing on Java projects specifically. Consideration will be given to the nature of the type of refactoring in the commits, specifically whether each commit is considered clean or dirty as not to generalize the impacts of all refactorings. The main priority of the first milestone is to find projects that have a high test coverage, in order to have a good set of projects to analyze in future milestones.

As a single commit could have many changes, refactoring may be only one of them, and as such we want to isolate these specific changes related to refactoring. In such cases, we will make a separate commit with only these refactoring changes and then evaluate running the test on this new commit.

If the test was passing before but now it fails, we need to understand why. On top of refactoring, there can be other changes such as modified or new functionality or logic changes, or even a bug [2] causing a test to fail. A reason why test code is modified can be in the event where a method is moved to another class, and a third party was calling that method in the original class, which could cause a compilation error due to no longer having access to the migrated code. This can be fixed by putting a delegate method in its place. A delegate method, however is not mandatory. If the delegate method is not made and the developers want to test the functionality, they need to instantiate the class where the new method location is, in the test. Due to this, we will need to distinguish between between compilation errors and test passes and failures. Additionally, clean refactorings can be performed with changes to the call site in the cases of refactorings such as Extract and Move Method refactoring types. This combination of changes may result in test failures, obscuring the effects of the changes in isolation. While a clean refactoring may in isolation, cause no change to functionality or resulting test failure, a change to the call site may be the cause and as such, where applicable, we test a refactoring in isolation by applying the refactoring with the original call site and observing the results of the tests.

We will conduct up to four types of testing to the source code in order to obtain results for our analysis. First we include tests at current commit including original refactorings and changes in order to determine if the commit, including the refactoring, causes test failure(s). Secondly, we include tests at previous commit not including refactorings and changes to verify and ensure that potentially failing tests were not already failing prior to commit including refactoring. Thirdly, we include the current commit with tests from previous commit, including the refactoring and changes to ensure that the refactoring did not break/change functionality given changes to tests. Lastly, we include tests at previous commit with the refactoring from next commit but call site of previous commit to ensure that given potentially failing tests alongside a clean refactoring, that the tests failed due to changes to call site, and not due to the clean refactoring.

In each case, we note the distinction between whether each instance of a failed test due to refactoring is a clean or dirty refactoring. Here we try to distinguish between the two. With the techniques in recent years, it is difficult to separate which code changes will affect the external functionality and which will only affect the internal functionality. Deciding if the commit is dirty or clean depends on whether the commit contains the code changes that will affect external functions. We cannot find a tool suitable to automatically distinguish between the two, therefore, we think detecting dirty or clean commits by manual analysis is a more suitable way then auto-detection. We have also constructed a ruleset based on several heuristics used to determine whether each refactoring instance is clean or dirty. The current cumulative set of heuristics can be found in the Findings section of the paper.

As not to generalize the effects of refactoring on the outcome of the corresponding tests to the refactored code, it is imperative that the distinction between clean and dirty refactoring is made. It is our intuition that dirty refactoring is more likely to result in failing tests, than clean refactoring since it may alter the functionality of the source code. To simply state that refactoring induces failing tests, would be incorrectly grouping two separate and distinct types of refactoring to the outcome of one type.

2 RELATED WORK

Rachatasumrit et al. [5] investigate the impact of refactoring edits on regression tests using the version history of Java open source projects in order to determine if (1) There are adequate regression tests for refactoring in practice, (2) How many existing regression tests are relevant to refactoring edits and this need to be re-run for the new version, and (3) What proportion of failure-inducing changes are relevant to refactorings. The results of the study suggest the need for new automated regression test augmentation and section techniques for validating refactoring edits, which could allow for the potential opportunity of saving regression testing cost. This study is similar to our study in that examines the impact of refactoring on tests. Furthermore, this study uses a tool (REFFINDER) to identify the types and locations between consecutive releases as ours will. It differs from ours in that it focuses specifically on regression tests while we focus on unit tests. We conduct our research with a consideration on the types of refactorings, specifically clean vs dirty refactorings in mind so as not to generalize conclusions. We build upon on their study by adding the distinction between clean and dirty refactoring instances, helping to prevent our results from generalizing the effects of refactoring. Additionally, we plan on using more projects which may assist in reducing bias based on specific projects that were selected.

Bavota et al. [2] examine the relationship between refactoring and bugs. As automated refactoring activities may cause bugs. The authors seek to answer three research questions in detail: (1) what extent or type of refactorings induce bugs xes, (2) whether specic kinds of refactorings induce more bugs than others, and (3) whether bugs occurred more in the source or target source code components. Furthermore, they also want to know what types of refactoring are more likely to cause bug fixes. They discovered that most affected files from refactoring types have no signicant difference comparing to files that have not been refactored. There are significant differences only some types, usually related to class hierarchy and have a high chance of inducing bugs. The authors mention they cannot find a study that is aimed at whether frequently refactorings in software systems during evolution will induce bugs. Our study is to find out when after refactoring, if refactoring will impact the test cases or not, and if not changing test cases after refactoring, if the system performance is impacted or not. The study of this paper is related to refactoring types and bug fixes. Both of our study and this paper are focused on the possibility of refactoring causing performance regression, bugs or even errors after refactoring. In this paper, they are only concerned with whether the refactoring will cause for future bug fixes and what types of refactoring will cause more bug fixes than others. Our study is focused on the relation between refactoring changes and test cases changes. The issue report of the related commit may include changes that are not related to refactoring and this may cause the refactoring unrelated bug fixes count as refactoring-related. We study the impact from refactoring types and which types of refactoring types will more easily to cause test cases errors.

Tsantalis et al. [8] implement a refactoring detection tool that can work without code similarity thresholds and without requiring fully built versions of system. Previous refactoring detection tools have many limitations including depending heavily on user-provided similarity thresholds, low accuracy, and requiring fully built software projects. This paper designed, implemented, and evaluated a new tool called RMiner (RefactoringMiner) to address the above-mentioned challenges. RMiner can detect refactorings with high accuracy at the commit level and does not require user-provided thresholds. The paper contains two research questions: (1) What is the accuracy of RefactoringMiner and how does it compare to the previous state-of-the-art and (2) What is the execution time of RefactoringMiner and how does it compare to the previous state-of-the-art. RefactoringMiner achieves very high precision, and has much better recall than RefDiff, while being up to 7 times faster. The study provides a tool that can analyze refactoring data without code similarity thresholds and also fully built versions. The result also showed that computing resource costs are less than previous tools. Furthermore, the tool may be very helpful for researchers who want to study the refactoring fields and improve the refactoring operations. Both our study and the paper are related to refactoring analysis. The paper and our study both extract data from the commit history from the version control system. Our study uses RefactoringMiner to find refactorings applied on the projects to be analyzed.

Elish et al. [3] proposed a classification of refactoring methods based on their measurable effect on (i) internal quality metrics as well as (ii) the effort it takes for testing the software, which is an external quality attribute. Software testing is an important activity in any software development and takes between 30 to 50 percent of the total effort spent on the development of the software. The effect on refactoring methods on software quality may vary, therefore, the goal of the paper is to help software developers decide which refactoring methods to use to optimize a software system in regards to software effort. The refactoring methods they study are: encapsulate field, extract method, hide method, consolidate conditional expression and extract class. We learn that different types of refactorings can have different outcomes on traditional object-oriented metrics. Similarities of the study are that both the related work as well as our study are evaluated on Java projects and see the effect of refactorings. Differences are that although the study shows that there is some impact on software testing caused by refactoring, this is only one small scale study. Due to this, we are not sure that refactoring even has any effect on software testing, and whether or not it increases or decreases the testing efforts. Our main goal is to see if refactoring even requires modification to tests.

We use existing unit tests rather traditional metrics to see if the proper and expected functionality of the code is still there after applying refactoring methods.

Silva et al. [6] explore and determine the motivations behind specific refactoring operations applied by developers. As there are a limited number of studies empirically investigating this topic, this motivation of this study is to help explore a relatively unknown area and fill gaps in missing knowledge in the area of refactoring by determining the actual motivations behind specific refactorings. This study is similar to ours in that it focuses on refactoring and utilizes a tool: RefactoringMiner [8] to obtain instances of refactorings from open source GitHub projects. Our study will also be doing this as a preliminary step, prior to analyzing the effects on tests associated with the refactored Java classes in question. This study differs from ours significantly in various ways. While our study is investigating the effects of refactoring, specifically on tests, this study covers a different area with its focus being on the motivations of specific refactoring types. Formally, the difference can be expressed as this study being concerned with the cause of refactorings, while ours is concerned with the effects of refactorings. Additionally, this study differs from ours in the methodologies employed. While this study contacts developers to obtain the cause of refactoring to be labeled and used for analysis, our study requires no correspondence with developers, and as such we will also not be concerned with the type of IDE used by developers for their refactoring instances. As our study is distinctly different, we will not be looking at improving upon this study but rather, to do as this study has done, by filling a missing gap in the research on refactoring and making a contribution by exploring an area with limited research.

Tsantalis et al. [7] obtain multi-dimensional findings through empirical analysis in order to answer the questions of (1) Do software developers perform different types of refactoring operations on test code and production code, (2) Which developers are responsible for refactorings, (3) Is there more refactoring activity before major releases than after, (4) Is refactoring activity on production code preceded by the addition or modification of test code, and (5) What is the purpose of the applied refactorings. This study is loosely related to ours is mostly similar in terms of the topic of refactoring alone. Conversely, the study differs in many ways. Our study will be more narrow and be looking at the effects of refactoring source code on the corresponding test classes, where this study is more broad and multidimensional, exploring many areas of refactoring. As this study is distinctly different from our study, we will not be aiming to improve on any particular features of the study. The only area where our study is likely to improve upon is the number of projects that we will be obtaining our data from which should help minimize the project bias that could be introduced when dealing with fewer projects.

Passier et al. [4] investigates the various ways in which refactorings can impact the API coverage of unit tests. They present an approach to tracking modifications of refactorings and analyzing their influence on the existing test suite. Finally, they provide insight and advice for how developers should update their test suite to migrate it, as well as discuss the current state and future potential of their Eclipse IDE plug-in for assisting developers with this challenge. This study is similar to our work in that it focuses on the impact of the changes in the program source code on unit tests. Where this study differs from ours, is that it looks to provide a solution to assist developers in facilitating the corresponding changes to test code where necessary to prevent broken alignment through the providing of advice to developers in the form of a tool where our study looks to investigate the impact with an emphasis on whether the refactoring is clean or dirty. The limitations of this study are that it provides a conceptual idea, and implementation of a tool but does not provide any empirical data on the effectiveness of the tool.

3 MOTIVATION

Refactoring is known to have positive benefits to software quality [6], however it is possible that a refactoring commit can contain other changes, e.g., those that introduce new logic or functionality. Currently, little research has be conducted to determine the effects of refactoring on test cases. Due to this, it is important not to mistakenly attribute test failures to refactoring when actual causes are additional changes with the refactoring (dirty refactorings or changes to call sites around clean refactorings). Our intuition is that clean refactorings will induce little or no failing test cases and that dirty refactorings or clean refactorings with changes to the call site will be more likely to induce test case failure.

We explore this topic by proposing the following research questions:

- **RQ1: Can clean refactorings alone induce test case failures?**
- **RQ2: Are dirty commits more likely to induce test case failure than clean commits?**
- **RQ3: Can changes to the call site around clean refactorings cause test failures?**
- **RQ4: Can rules that are defined to distinguish between clean and dirty refactorings be applied programmatically to identify other instances?***

***RQ4** will studied in our future work.

4 DATA SELECTION

We select our target to be open source projects available on GitHub with over 70% of code coverage. We choose projects of medium size since smaller projects little refactoring instances and larger projects showed too many instances to analyze. Another criteria we add is that the project must be build-able and testable, as sufficient test coverage is required and our the purpose of the project is to check the impact of refactorings on unit tests. For the purpose of the course, we focus on the *commons-rng* and *commons-lang* project as our subject.

The data collection process consisted of obtaining commit hashes between releases, which included identified cases of refactorings in files that also had test coverage such that we could analyze the effects of refactoring on the outcome of test cases.

5 APPROACH

The steps of our approach are shown in Fig.1.

The general approach consisted of first running the RefactoringMiner [8] tool and obtaining all refactoring instances between two releases, in our case, between two different releases of the *commons-rng* and *commons-lang* open source project. After filtering out uninteresting instances that should not affect functionality such as renaming of attributes, methods, and classes, the data was parsed in order to have easy access to the commit hash, and file name and location. The second step was obtaining the test coverage for each test file in every package of the project. The coverage was done at the time file level to be able to view more details such as which files were covered by tests and the level of such coverage. Upon obtaining the coverage, the data was parsed into a list of source code files names that had test coverage and would be considered to identify candidates for analysis. The intersection between file names in this step and the previous file names in the parsed results of RefactoringMiner [8] step, allowed for the elimination of commit hashes that contained refactorings but had no test coverage, and thus could not be used in this study. After omitting the non-intersecting commit hashes, we obtained the following list of commit hashes containing refactorings on files that had test coverage between releases.

Due to time constraints and the limited number of intersecting commit hashes in our approach, for the purpose of this study, we have elected to perform manual analysis on these commits on whether a commit is clean or dirty, along with a negotiated agreement approach. We have utilized the RefactoringAware [1] extension for Google Chrome to assist in identifying the refactoring instances among the changed lines of code and view the mappings between these changes between the two states of the file, before and after the commit. Our manual analysis consists of the following steps:

- 1. Checkout the commit using the **git checkout** command
- 2. Build the project using **mvn clean**.
- 3. Run the tests to obtain the pass/fail result following the refactoring and other possible code changes using **mvn clean test**.
- 4. Check out the parent commit (parent hash obtained from commit on GitHub) and **mvn clean test** to obtain pass/fail results.
- 5. Open the commit in Google Chrome and identify the refactoring using the RefactoringAware [1] extension.
- 6. Manually inspect the refactoring and determine whether the commit is clean or dirty.
- 7. If test code was changed at the same time as the commit, we check out the parent commit and make a copy of the test file.
- 8. Checkout the commit child commit, or commit under investigation and replace the test file with the copied test file from the parent commit.
- 9. Run tests and observe pass/fail outcome.

The logic behind our approach involves the outcome and results of the tests over up to four states of the source code files which have test code coverage. The first test run determines if the refactoring, and/or any changes in the commit caused the tests to fail. In the event that the tests had failed, it is possible they were failing before the commit containing the refactoring. By checking out the parent commit, we have a before and after state and can allow us to view any pass or fail as a result of the code changes and not simply inherited passing or failing tests. By running the third set of tests which include the test file(s) before the refactorings, in the case of changed test file(s), we can ensure that the functionality was not changed but hidden by additional changes in the test file. As the functionality should not change, the parent tests should pass even with the refactoring. Finally, the fourth set of test cases will clarify the effects of clean refactorings when additional changes are made to the call site of the refactoring such as changes to conditional logic which may affect the test cases, and functionality of the code, despite the refactoring itself being a clean refactoring in isolation.

It is worth noting that additional benefit to running the tests before and after is that we identify some tests that are flaky. This is important as flakiness can hide bugs and possibly functionality breaking changes in the code. With the **mvn clean test** command, failing tests are re-run on this project and we can see a resulting record of which tests are flaky. We will not be able to draw any conclusions about the effects of clean or dirty refactorings from tests that are covered by flaky tests. As such, in the event of finding instances of such refactorings, we will make notes of the instances but the refactorings will be omitted from our analysis.

We define a clean refactoring commit as a commit that does not have tangled changes. A clean refactoring commit only has line additions and removals that relate to the parent commit and the logic can be completely mapped. No new logic is added, nor is any logic removed in the child commit. In the split view of the changes on GitHub, we should be able to map all changes, which contain the same logic from the parent and child changes to each other. If this is not possible, then the commit is dirty. The child commit either contains more code logic or less code logic or different modified logic compared to the old commit. Cases of both are found are found through our examined commits. Current rules can be found in Findings section of the paper. Upon further investigation of more cases in the upcoming and future milestone, there may be more rules defined and more refined, and when there is sufficient amount of rules, we can go from manual investigation to automation to detect adherences and violations of these rules.

For some of the commits that were analyzed, we noticed that not only do they include a main refactoring, but they also include new logic around them such as changes around the call sites. An example of this that we found upon manual analysis is in commit **65abc5dcfe4198b4244d4083d790ea870edbebe5**. This is an extract and move which changes a number of items. The conditional **!=** becomes < and **IllegalArgumentException** becomes **IllegalStateException** in the **checkStateSize()** method. The **setStateInternal()** method changes from throwing **UnsupportedOperationException** to checking the length and then throwing **IllegalStateException**. It is also dirty because it is an extract and move that changes the number of items. Additionally, there's a check before it shows the method, in the set method: instead of automatically throwing the exception, it checks if the **state.length != 0**.

We perform further manual analysis: for each commit that we analyzed, we isolate the refactoring and create a commit from it, and run the tests on this new commit. The goal of this manual analysis is to only test on the refactoring itself, by removing any tangled changes. The Refactoring Aware extension helps us find the targeted code to use in the new commit. We do this by checking out of the parent commit, and add in only the core refactors from the refactored commit. Below, we write the steps we do at each parent commit, to put add in the code of the isolated refactorings. We show the results of the tests before the changes and the results of the tests after the changes. During this second manual analysis step, we saw instances of refactorings where when isolating by removing changes around the call sites, the remaining core refactorings was just to change the type of a variable e.g., from type **int** to type **long**. We choose to neglect these refactorings in the future as they are trivial changes.

6 FINDINGS

Our current findings are listed as follows:

- For some of the commits that were analyzed, we noticed that not only do they include a main refactoring, but they also include new logic around them such as changes around the call sites
- Purely clean refactorings did not cause any failures
- Dirty refactorings analyzed so far did not cause immediate failures on current commit, however they did change functionality (unable to pass tests from previous commit due to changes)
- Clean refactorings with changes to call site did not cause immediate failures on current commit, however functionality has changed. Clean refactoring alone does not induce failure either

Some of the defined rules will need to be discussed and may be affected by the development stage of the tool. We may have cases that we will need to initially identify as dirty but manually inspect after to ensure that it is not a false positive. As we go over more refactoring instances and build our ruleset, as well as familiarize ourselves with the RefactoringMiner API [8], we will expand and refine our ruleset. Rules we defined so far are listed as follows.

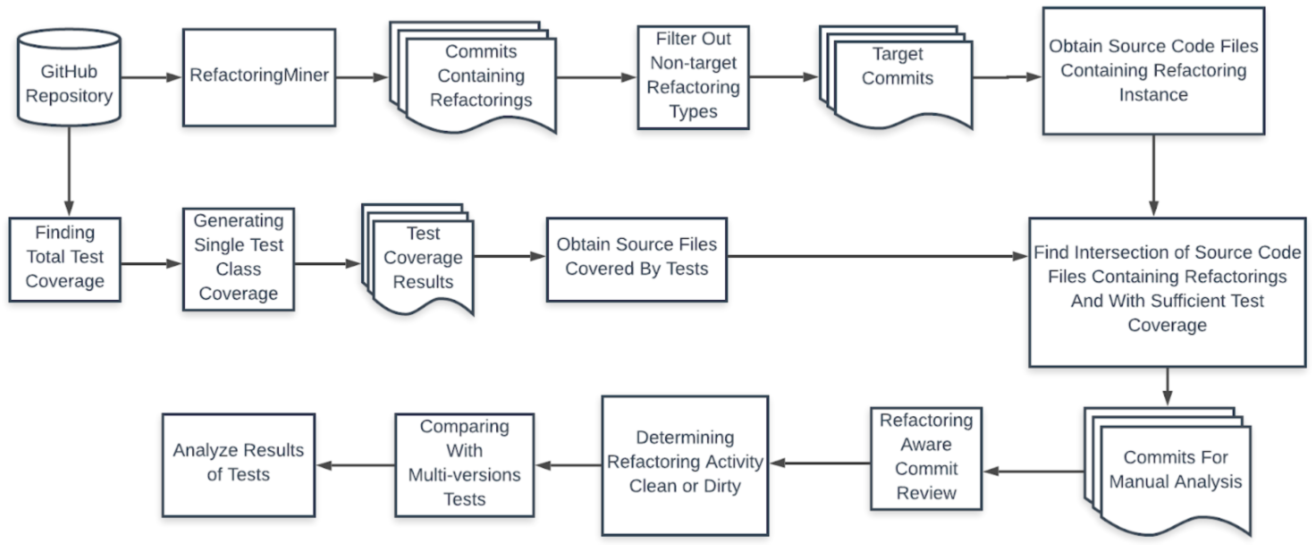
- R1: Extracted Method contains no additions or subtractions or modifications to the method body other than code extracted are considered as clean
- R2: Moved Method contains no additions or subtractions or modifications to the method body other than the code that was moved are considered as clean
- R3: Extracted Method contains additions are considered as dirty
- R4: Extracted Method contains subtractions are considered as dirty
- R5: Extracted Method contains modifications to logic are considered as dirty
- R6: Extracted Method contains modified conditional logic are considered as dirty
- R7: Extracted method makes call to the super class (need to further define this rule)
- R8: Moved Method throws new type of exception are considered as dirty
- R9: Moved Method prints or logs modified message are considered as dirty
- R10: Moved Method contains new or modified conditional logic are considered as dirty
- R11: Delegate method is created and called, where the old method is deprecated but called in the delegate method is considered as clean
- R12: Iterating logic of for loop changed from **(for int i=0; i<10; i++)** to **(for int i=10; i>0; i–)** Body not changed and no reference to i in body (for now we consider dirty as we do not have a viable system to detect the equivalence of the logic)
- R13: Extracted method contains reordered lines (dirty for now as it may impact the logic but we will need to refine as a line can be potentially moved higher or lower in the block depending on if/when it is referenced)
- R14: Changes to an attribute type solely is not considered as a dirty refactoring

While our initial intuition was that refactoring instances of type Rename Class should be included in our study, upon further consideration, we feel this should not be included. Our initial thought was that if the class was renamed manually, that it was possible that subsequent object creation with respect to this class could cause compilation errors and could be an instance of a clean refactoring that could cause failures. Use of refactoring tools and IDE support will typically update occurrences and during our isolated testing process. Additionally, it is difficult to try to apply only this change if we do not consider all object creation occurrences as part of the refactoring. For this reason, we will consider omitting this type of refactoring.

Changing types is another case that can be problematic. When attempting to apply only the refactoring in isolation, there can be other changes that propagate with this change. This can affect code from assignments to operations, and even subsequent passing of parameters into method calls. For example, changing the type of a variable may cause this variable to be passed into a method call with no longer is accepting the correct type. This indicates that some of the refactoring instances cannot be investigated in such strict isolation, but rather some may require logical grouping as they are likely to occur at the same time. Another problematic aspect of the changing of types is that in the case of objects, the require parameters can change as well, which can make detection difficult. Other side effects could include common methods such as a **toString()** method which now returns a significantly different output, possibly failing a test, but yet is still considered as a clean refactoring. Furthermore, with primitives, the isolated changes may require casting to simulate. For the moment we will consider these changes in types to be clean in isolation but we will need to develop our detection methods in order to automate this and detect variances.

Other instances of refactorings can be extremely difficult to automate detection for. One such case is the ternary operator being refactored to or from an ifstatement. Not all of this logic is always straightforward and can be missed by the tool when additional refactoring takes place at the same time, such as substituting the contents of an expression within the ternary operator with an extracted attribute. Furthermore, there was a case observed which transformed an ifstatement to an if statement with a break, which may be difficult to automate, despite being logically equivalent statements. This will also be the case for situations including for

Figure 1: Approach to study the impact of refactoring on test cases


GitHub Repository
RefactoringMiner
Commits Containing Refactorings
Filter Out Non-target Refactoring Types
Target Commits
Obtain Source Code Files Containing Refactoring Instance
Finding Total Test Coverage
Generating Single Test Class Coverage
Test Coverage Results
Obtain Source Files Covered By Tests
Find Intersection of Source Code Files Containing Refactorings And With Sufficient Test Coverage
Commits For Manual Analysis
Refactoring Aware Commit Review
Determining Refactoring Activity Clean or Dirty
Comparing With Multi-versions Tests
Analyze Results of Tests

loops which increment from **i=0** to **n** while **i<n** being transformed to decrementing from **i=n** to **0** while **i>0**. Further consideration will need to be given to how to handle such situations.

As mentioned, there are cases where the Refactoring Aware tool does not find all refactoring occurrences. The common observation is that when there are combinations of refactorings, sometimes the tool will see the differences as too distinct to view as a specific refactoring. This is often when renaming is combined with other types. There was even one such case where an attribute was recognized as both moved to another class and in the same commit, also in the same class as a changed type. This seems to be a contradiction where both cannot occur simultaneously unless the attribute is moved to another class where the type is also changed. In this particular case, the attribute was moved to another class and had the type remain the same in the new class, yet also within the originating class, the attribute was indicated as having its type changed. Investigation seems to suggest that the supposed changed type is actually a new attribute, unrelated to the attribute in the originating class.

One final point worth noting is that we other cases where the refactoring was not recognized by the tool. This particular case had a method called **sample()** which called a method **nextPoisson()** in the parent commit. After the refactoring took place, the **nextPoisson()** method was removed and a portion of its contents was moved directly to **sample()**, which previously called **nextPoisson()**. The portion that was moved from **nextPoisson()** to **sample()** was part of the else of a conditional statement in the previous method, and had some additional changes. As the tool did not pick this up and identify as a refactoring instance, it should be then concluded that some cases of refactorings may be so dirty that they are not even identified as refactorings altogether. Further thought and consideration will need to be then given to this issue as a significantly dirty refactoring will inherently include a significant number of changes which will increase the probability of functionality altering changes. These are exactly the types of changes that may induce test case failure and thus there is an important potentially unseen threshold that separates an extremely dirty refactoring from an unidentified refactoring.

Out of the total 16 refactoring commits analyzed, we found 41 refactoring instances. Upon manually analyzing these instances with the help of the RefactoringAware [1] extension, we found that 15 of these instances were clean, while the remaining 17 were dirty. From the 24 instances that were clean, we found that only 2 of these instances had dirty call sites.

For each of the 16 refactoring commits analyzed, we run the unit tests at those commits, resulting in 12 commits with passing tests and 4 with failing tests. Next, we run the tests on their parent commits. For the 12 child commits that had passing tests, 1 of these had a parent commit that had a failing test, and the remaining 3 with passing tests. For the 4 child commits with failing tests, all of their parent commits also had failing tests.

For the *commons-lang* project, of the four commits examined, there were a total of five refactoring instances. Each of these five refactoring instances were clean refactorings. Additionally, each of these clean instances were isolated already, and as such there was no need to apply the refactoring on the parent commit code as this application was already contained in the refactoring commit. No changes to any tests were present so there was no need to apply tests from the parent commit as well. While it should be noted that there were test failures at the refactoring commits, in all such cases, the parent commit had the same number of failures, errors, and skipped tests. This indicates that the clean refactoring did not induce further test failure in 100% of cases.

For the *commons-rng* project, there are nineteen instances of clean refactorings, and fourteen dirty refactorings. There is a case where for a refactoring commit, both the parent and child commit tests are passing, but when isolating the refactoring manually, this induced test failures for two of the three refactoring instances. There is also a refactoring commit where in itself had tests failures but when manually isolating the refactoring changes, it also induced failures but no new failures compared to the original commit, which we can say as passing as it did not induce new test case failures.

7 THREATS TO VALIDITY

Our approach does have some threats to internal validity based on the criteria of selection of candidates of refactorings to analyze. Our criteria requires refactoring instances to be in files that have test code coverage in order to identify the effects of the refactorings. In an ideal situation, the test code coverage would be 100% but in most cases this simply is not the case and it is generally much lower. Due to the nature of approach, any files that do not have test code coverage will be omitted from our study, regardless of whether they contain instances of refactorings. Projects with higher test code coverage will experience this effect less than projects with lower coverage but not every refactoring can be analyzed and included in our approach on this basis. It should be stated that these omitted refactorings could very well contain code breaking changes or major changes to the functionality but these changes simply cannot be detected due to the absence of test code coverage. Whether they contain these types of changes cannot be determined and as such while we are not intentionally losing instances of refactoring or purposely omitting them, our dataset will be a subset of the total list of refactorings as a result of our approach.

As our approach is obtaining the test code coverage at the beginning and ending of a release, the accuracy of the test code coverage may not be exactly the same throughout the release. It is possible that the coverage fluctuates with tests being removed, added and modified. Much like cases of files containing instances of refactorings but that do not have test code coverage mentioned, another such issue that should be noted is that some of these files have minimal test code coverage. Due to time constraints and limitations due to our approach not yet being automated, we currently have as a criteria only that files have at least some test coverage. It is possible that the potentially minimal test code coverage does not completely cover the refactoring instances identified. We intend on automating our approach and including a defined threshold of test code coverage of files. As a type of heuristic, files that have very high test code coverage at the beginning and end of a release likely have sufficiently high coverage throughout the release. Once the process is fully automated, we can consider for each commit containing a refactoring, to check test coverage at the immediate parent commit for each instance and possibly filtering out the refactoring as part of the omitted instance that had no test code coverage at all.

8 FUTURE WORK

In the future we plan on expanding upon our study in a number of ways. We plan on expanding our analysis with the inclusion of more instances by expanding our dataset to include instances from more releases as well as including other projects. Initially, we will continue our manual analysis while we automate the current approach. A vital part of our approach requires identifying whether a commit is clean or dirty and this is currently being done manually. In the future we will be automating this by using the Refactoring-Miner API [8] and examining the code changes and the mappings of these changes. These mappings will be essential and assist us with the rules listed later in this document, as well as future rules. The parallel manual inspection alongside the developing automated tool will give us more edge cases to investigate, to determine what type of modifications can be defined as rules to determine clean or dirty refactorings allowing for automating all steps of the project, leading to the development of an automated tool. We plan to automate the steps listed in our approach, shown in Fig. 1.

9 CONCLUSION

Our empirical study examines two GitHub projects: *commons-rng* and *commons-lang* and how the tests of these projects are affected by refactoring commits. Refactoring is known to have positive benefits to software quality, however it is possible that a refactoring commit can contain other changes such as introducing new functionality. Currently, little research has be conducted to determine the effects of refactoring on test cases. Due to this, it is important not to mistakenly attribute test failures to refactoring when actual causes are additional changes with the refactoring. Our approach examines refactorings done between two consecutive releases, analyzing the interesting refactoring commits and studying their impact on existing unit tests. For tangled refactorings, we create a further step by manually creating a new commit and isolating the core refactorings. We also define four types of manual testing, based on the nature of the refactoring and its test results. In total, we analyzed 16 refactoring commits, where each commit contains at least one one refactoring instance. Our main findings are that purely clean refactorings did not cause any failures and for some of the commits analyzed, not only do they include a main refactoring, but also include new logic around them such as changes around the call sites. We also include a ruleset established on criteria that determines a distinction between a clean and dirty refactoring. In the future, we plan to automate all steps and create a tool, to determine the impact of refactorings on existing source code unit tests, creating a test-aware tool.

REFERENCES

[1] [n.d.]. Refactoring Aware Commit Review. https://chrome.google.com/webstore/detail/refactoring-aware-commit/lnloiaibmonmmpnfibfjjlfcddoppmgd?hl=en

[2] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. 2012. When Does a Refactoring Induce Bugs? An Empirical Study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation.* 104–113. https://doi.org/10.1109/SCAM.2012.20

[3] Karim O. Elish and Mohammad Alshayeb. 2009. Investigating the Effect of Refactoring on Software Testing Effort. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference (APSEC '09).* IEEE Computer Society, Washington, DC, USA, 29–34. https://doi.org/10.1109/APSEC.2009.14

[4] Harrie Passier, Lex Bijlsma, and Christoph Bockisch. 2016. Maintaining Unit Tests During Refactoring. In *Proceedings of the 13th International Conference on*

Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16). ACM, New York, NY, USA, Article 18, 6 pages. https://doi.org/10.1145/2972206.2972223

[5] N. Rachatasumrit and M. Kim. 2012. An empirical investigation into the impact of refactoring on regression testing. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 357–366. https://doi.org/10.1109/ICSM.2012.6405293

[6] 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 (FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[7] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A Multidimensional Empirical Study on Refactoring Activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '13)*. IBM Corp., Riverton, NJ, USA, 132–146. http://dl.acm.org/citation.cfm?id=2555523.2555539

[8] 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 (ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206