DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2020

A study of Pseudo-tested Methods in an Android Continuous Integration Environment

JOHN KVARNEFALK

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

A study of Pseudo-tested Methods in an Android Continuous Integration Environment

JOHN KVARNEFALK

Masters in Computer Science
Date: July 6, 2020
Supervisor: Deepika Tiwari
Examiner: Martin Monperrus
School of Electrical Engineering and Computer Science
Host company: Spotify AB
Swedish title: En studie av pseudo-testade metoder i en continuous integration-miljö för Android

Abstract

As mobile devices become more capable and mobile adaption grows across the world, the complexity of mobile applications increases. For example, mobile applications let consumers view their banking details and legitimize themselves. With mobile applications serving these crucial functions, the need for quality and robustness increases. Unit tests are one way to measure the quality of an application. However, there is a need to evaluate the unit tests themselves in order to increase their effectiveness. Mutation testing is one approach to test the application source code by deliberately mutating it in an effort to see if the unit tests of the application detect the change.

This study investigates mutation testing on an Android application. More specifically, it evaluates a form of extreme mutation testing using pseudo-tested methods. With pseudo-tested methods, entire method bodies are replaced with a single return statement. This study introduces a framework for detecting pseudo-tested methods within the continuous integration environment as well as the capability of measuring the usefulness of generated reports that highlight these methods. Moreover, the study conducts interviews with multiple developers discussing the pseudo-tested methods in detail.

We implement this framework within the continuous integration environment using multiple components. Our results show that developers are positive towards having a tool for detecting pseudo-tested within the continuous integration environment. Moreover, the results indicate that methods written by developers are more useful to test than auto-generated methods. We draw the conclusion that detection of pseudo-tested methods can be beneficial in increasing the quality of mobile applications. However, we note that further work is needed to eliminate the prevalence of false-positives found in our study

Sammanfattning

Då mobiler blir mer kapabla och mobilanvändandet ökar runt om i världen blir mobila applikationer allt mer komplexa. Till exempel låter mobila applikationer användare visa sina bankuppgifter eller legitimera sig själva. När mobila applikationer hanterar dessa kritiska funktioner ökar vikten av robusta applikationer med hög kvalitet. Därför används enhetstestning för att bedöma kvaliteten av en applikation. Det medför att det dessutom finns ett behov av att utvärdera enhetstesterna av en applikation för att öka deras effektivitet. Mutationstestning är en metod där applikationens källkod medvetet ändras för att se om applikationens enhetstester kan upptäcka förändringen.

Denna studie undersöker mutationstestning i en Android-applikation. Studien undersöker specifikt en mutationstestningsmetodik som kallas pseudo-testade metoder. Med pseudo-testade metoder ändras hela metoder till att endast innehålla ett ensamt return-uttryck. Denna studie introducerar ett ramverk för att exekvera pseudo-testade metoder och samla in återkoppling gällande ramverket i continuous integration-miljön. Dessutom utförs ett antal intervjuer för att diskutera pseudo-testade metoder i detalj.

Vi implementerar ramverket med hjälp utav flera olika komponenter. Resultaten pekar på att utvecklare är positiva till att ha ett verktyg för att upptäcka pseudo-testade metoder i continuous integration-miljön. Däremot är det av stor vikt att metoder som är felaktigt rapporterade minimeras. Samtidigt indikerar resultaten att metoder skrivna av utvecklare är mer relevanta att testa än metoder som är autogenererade. Vi drar slutsatsen att pseudo-testade metoder kan vara användbara för att förbättra kvaliteten av mobila applikationer. Däremot föreslår vi att framtida arbete är av vikt för att eliminera falskt positiva rapporterade metoder.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Problem Definition . . . 2
1.2.1 Research Questions . . . 2
1.2.2 Objectives . . . 3
1.2.3 Limitations . . . 3
1.3 Ethics and Sustainability . . . 3
1.4 Outline . . . 3

2 Background **5**
2.1 Terminology . . . 5
2.2 Preliminaries . . . 6
2.2.1 Control Flow Graph . . . 6
2.2.2 Software Quality Metrics . . . 7
2.2.3 Continuous Integration . . . 9
2.3 Tooling . . . 10
2.3.1 The CI Server . . . 10
2.3.2 GitHub Checks . . . 10
2.3.3 Gradle . . . 10
2.3.4 Android Development Environment . . . 11

3 Related work **13**
3.1 Automated Fault Detection Tools in CI . . . 13
3.2 Mutation Testing . . . 15
3.3 Pseudo-Tested Methods . . . 17
3.4 Mutation Testing on Android . . . 18

4 Methodology **21**
4.1 Experiments in CI . . . 21
4.1.1 Technical Infrastructure . . . 21

4.1.2 Pilot Study 26
4.1.3 Feedback Collection 27
4.1.4 Filtering of Reports 27
4.1.5 Reporting Format 27
4.1.6 Experiment Duration 29
4.2 Interviews 30
4.2.1 Subject Selection 30
4.2.2 Response Collection 30
4.2.3 Response Compilation 31

5 Results **32**
5.1 RQ1 - Usefulness of Presenting Pseudo-tested Methods in the CI 32
5.2 RQ2: Developer Interviews 35
5.2.1 Background of Interview Subjects 37
5.2.2 Testing Weakness 37
5.2.3 Intent to Correct Weakness 38
5.2.4 Report Context 38
5.2.5 Tool Relevance Across Project 39
5.2.6 Key Takeaways 40
5.3 RQ3: Developer Actions 40
5.4 RQ4: Classification of Pseudo-tested Methods 41

6 Discussion **43**
6.1 Threats to Validity 43
6.1.1 Robustness of the Tooling 43
6.1.2 Limited Scope of the Study 44
6.2 Auto-generated Methods 45
6.3 Fault Presentation 46
6.3.1 Structure of the Report 46
6.3.2 Bug Ticket 47
6.3.3 Status of Checks 47
6.4 Correcting a Fault 48
6.5 When is a Method Change Relevant? 48

7 Conclusions **50**
7.1 Usefulness of Pseudo-tested Methods 50
7.2 Developer Action on Reported Faults 50
7.3 Relevance of Presented Methods 51
7.4 Future Work 51

Bibliography **52**

A Transcribed Interviews **57**
A.1 Interview 1 57
A.2 Interview 2 61
A.3 Interview 3 67
A.4 Interview 4 71
A.5 Interview 5 75

Chapter 1

Introduction

1.1 Background

Over the last decade, mobile applications have become crucial to our everyday life. There are millions of applications available to download on the Google Play Store [1] and the Apple App Store [2]. Mobile applications not only provide leisure and entertainment, many are also responsible for critical activities such as health monitoring, banking, and digital identification. Therefore, it is important that the quality of mobile applications is high to ensure a robust and reliable experience. One way to do this is to make sure that they are tested thoroughly. An emerging research area is the application of mutation testing in the mobile application domain. Manifestations of interest in this area include a wide range of papers introducing custom mutation testing frameworks and mutation operators for Android [1, 2, 3, 4, 5, 6, 7].

Despite the wide interest, this research area still has issues left to address. Even though multiple frameworks have been introduced, existing literature lacks the perspective of the developer and of the development lifecycle. The focus has been directed towards improving runtime and crafting effective mutation operators. Research into the actual integration of these tools into the developer workflow and environment is not explored to the same extent.

Parallel to the research on mutation testing on mobile applications, mutation testing in general has also been extensively studied. A new way of performing mutation testing, called extreme mutation testing, has arisen where instead of regular mutation operators, the source code of an application is mutated at a higher level to reveal other types of faults [8, 9]. A specific branch

[1] https://play.google.com/store
[2] https://apps.apple.com/us/genre/ios/id36

of extreme mutation testing that is of special interest to this thesis is that of pseudo-tested methods [8]. A recently developed framework called Descartes [9] detects pseudo-tested methods by replacing entire method bodies with a single return statement.

Furthermore, with mobile application projects growing larger in size as well as in the number of developers involved, the importance of having robust continuous integration (CI) practices is growing [10]. Research has been conducted into how software quality tools, such as line coverage and mutation testing frameworks, can be integrated into CI systems to interact with developers in a useful manner [11, 12].

In this thesis, we explore the combination of extreme mutation testing and CI in the Android application development environment. We utilize the concept of pseudo-tested methods in a large-scale Android application. The mutation testing we perform is made visible to the developers in a CI system. Feedback on the usefulness of the tool is collected from the developers during the course of the study. Finally, interviews are conducted with developers to assess their opinions about the usefulness of the tool.

1.2 Problem Definition

To investigate and evaluate the integration of detecting pseudo-tested methods in a CI system, we formulate four distinct research questions.

1.2.1 Research Questions

RQ1: According to the developer feedback collected, what is the perceived usefulness of presenting pseudo-tested methods within the CI system?

RQ2: What can be concluded about developers' opinions on the highlighted pseudo-tested methods, judging by the interviews?

RQ3: To what extent do developers take actions on improving their tests after being presented with the report on pseudo-tested methods?

RQ4: What is the feasibility of classifying pseudo-tested methods as interesting or irrelevant?

1.2.2 Objectives

To answer these research questions, our study focuses on a number of objectives.

The first objective is to design a complete architecture for detecting pseudo-tested methods within a CI system that is also able to collect developer feedback. The system should be designed to be robust and capable of running under the constraints of the CI system. Another important goal is to provide a clear and intuitive interface for developers to give their feedback of the system.

The second objective is to conduct developer interviews to discuss and collect opinions about the tooling. The main goal is to find ideas for improvement, as well as to discuss the pseudo-tested methods reported and the reason why these methods were pseudo-tested.

1.2.3 Limitations

This thesis focuses only on evaluating the concept of pseudo-tested methods in a CI system. We do not focus on developing a new mutation testing framework. The thesis only evaluates the tooling from a usefulness perspective. Thus, we do not focus on evaluations such as runtime comparisons of the tool.

1.3 Ethics and Sustainability

The tooling examined in this thesis could be used to improve the quality of mobile applications. Given that mobile applications are used widely for increasingly critical purposes, an improvement in quality could lead to a positive outcome on societal sustainability. Mobile applications lacking in quality can lead to security vulnerabilities. If security vulnerabilities are present in a mobile banking application or a digital identification application, the consequences can be severe. With a framework responsible for improving the quality of mobile applications, some of these vulnerabilities could be detected before a potential exploit. Thus, the results from this thesis could have a major impact on societal sustainability.

1.4 Outline

The rest of the thesis is organized as follows: Chapter 2 gives an introduction to software quality metrics, tooling used in the scope of this thesis, as well as a background of the technical infrastructure of the project where the tool was

implemented. Chapter 3 introduces closely related work. Chapter 4 presents the methodology used in this thesis. We introduce the method used to integrate the tooling in the system as well as the collection of developer feedback. Furthermore, we present the methodology used to conduct and analyze our interviews. Chapter 5 presents our key results from the feedback collection and from the interviews. Chapter 6 discusses how our results relate to our research questions. Chapter 7 concludes the thesis with a summary as well as potential future work in this line of research.

Chapter 2

Background

2.1 Terminology

This section introduces the terminology used in the context of this thesis. A basic understanding of software development and a general understanding of computer science is expected of the reader. In Table 2.1 follows a list of some of the terms that will be used throughout this thesis.

Term	Description
Test suite	A collection of test cases, often used to determine the fitness of an entire program.
P25, P50, P75	When measuring performance of a system, it is common to measure it at different percentiles. P25 is the top 25th percentile, P50 is the 50th, and P75 the 75th.
ETL	Extract-Transform-Load is a procedure where data is extracted from a system, transformed to fit the requirements of a destination system, and finally loaded into the destination system.
DSL	A Domain Specific Language is a computer language specialised to work with a particular application domain.
Android APK	Android Application Package is the package file format used by the Android operating system for distribution of Android applications.

Table 2.1: Terminology used within the context of this thesis

2.2 Preliminaries

2.2.1 Control Flow Graph

To reason about software programs and perform analysis, a control-flow graph (CFG) is often constructed [13]. A control-flow graph uses graph notation to represent all the paths that might be traversed during program execution. In a control-flow graph, each node represents a basic block. A basic block is a code statement without any jumps or jump targets. The edges in the graph represent the jumps in the control flow. Consider the code in Listing 2.1. The code represents a simple function that calculates the absolute value of its input argument with accompanying test cases. Fig 2.2 is an example of a CFG for the `abs` function in Listing 2.1. In this CFG, each line of the function corresponds to a node in the CFG. The transition between lines is represented by an edge, and the if conditional statement produces two outgoing edges.

```
1  public int abs(int val) {
2      int a = val
3      if (a <= 0) {
4          a = -1 * a;
5      }
6      return a;
7  }
8
9  @Test
10 public void testAbsPositive(){
11     int test = abs(1);
12     assert (test == 1);
13 }
14
15 @Test
16 public void testAbsZero(){
17     int test = abs(0);
18     assert (test == 0);
19 }
```

Figure 2.1: Simple method calculating the absolute value with test cases.

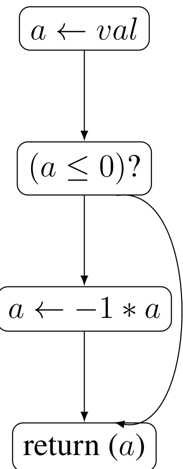

Figure 2.2: Control Flow Graph of abs method in 2.1

2.2.2 Software Quality Metrics

In software development, testing is used to give developers confidence that the program behaves according to expectations. Software tests are specified by writing test cases that test the functionalities of a program. The entire collection of these test cases constitute the test suite of a program. There exist multiple metrics to understand the quality of a program with respect to its test suite. We discuss some of these metrics below.

Let us first define a set of variables that we use to formally introduce the software testing concepts below. For a given program under test, T is the test suite. G is the control-flow-graph representing the program. N is the collection of nodes in G, and E is the collection of all edges in G. $path(T)$ is a generator function that returns a set of all paths traversed in G as a result of executing T.

Statement Coverage

Definition 2.2.1. Test suite T satisfies statement coverage on graph G iff for every syntactically reachable node n in N, there is some path p in $path(T)$ such that p visits n.

One way of assessing the quality of a test suite against its program is to measure the statement coverage. Looking at the example in 2.1, complete statement coverage implies lines 2-6 are executed by the test cases. A program with a high statement coverage gives the developer confidence that the test cases are covering a high percentage of the code. However, the metric does

not give any information about what the test suite actually tests. A statement is defined to be covered when a test case exercises the statement. Statement coverage does not have any requirements on the statement actually being asserted. Thus, in theory, we could declare a test suite without asserting anything in the entire program. We would achieve full statement coverage, but in practice, we would not test our program. Even though the statement coverage metric does not give any information on what the test suite actually tests, it can still serve as an indicator for poorly tested parts of the code. The statement coverage metric is widely used in the industry due to its simplicity. A large majority of existing testing frameworks have integrated support for the metric [12]. It can be regarded as a baseline for evaluating the test suite of a program.

Edge Coverage

Definition 2.2.2. Test suite T satisfies edge coverage on graph G iff for every syntactically reachable edge e in E, there is some path p in $path(T)$ such that p traverses over e.

Edge coverage is a superset of statement coverage. Looking at the example in 2.1 again, complete edge coverage implies that lines 2-6 are executed and the if-statement on line 3 is evaluated to both `true` and `false`. Edge coverage is similar to statement coverage in that it does not give information about what the test suite actually tests. However, edge coverage is always at least as good an indicator of test quality as statement coverage. If a test suite has edge coverage for a program it implies that it also has statement coverage. The implications are due to trivial graph theory, a traversal of all edges in a graph is also a traversal of all of the nodes.

Mutation Testing

Mutation testing is a concept used to evaluate the effectiveness of software tests. The concept of mutation testing is to insert minor faults into programs and see if the test suite detects the fault. These minor faults result in a modification of the original program, called a *mutant*. Demillo and Ooutt [14] originally proposed the method to measure the effectiveness of a test suite in detecting, and possibly "killing" these mutants. If at least one test case fails while running the test suite on the mutant, the mutant is marked as killed. Consequently, if the mutant is not killed, it is marked as a living mutant. The more mutants the test suite kills, the more effective it is. This effectiveness is measured in a metric called the *mutation score*, which is the ratio of killed

mutants over the total number of mutants. The mutation score is considered to be one of the best metrics for evaluating the efficiency of a test suite [15]. Evaluating the example in Fig. 2.1 using statement or edge coverage metric would give us full coverage. Consider changing the statement `return -1 * val` to `return 1 * val;`. This mutation is a unary deletion mutation since we remove the unary operator of the constant 1 on line 4 in Listing. 2.1. This would produce a mutant that is not killed by the test suite. Consequently, the mutation score metric of this program would reflect that the test suite can be improved. The mutation metric, therefore, contributes to more insight into the quality of the test suite. See Table 2.2.2 for a list of common mutation operators and their corresponding categories.

	Name	Scope
AOR	Arithmetic Operator Replacement	$a + b \to \{a, b, a - b, a * b, \frac{a}{b}, a \bmod b\}$
LCR	Logical Connector Replacement	$a \wedge b \to \{a, b, a \vee b, \top, \bot\}$
ROR	Relational Operator Replacement	$a > b \to \{a < b, a \leq b, a \geq b, \top, \bot\}$
UOI	Unary Operator Insertion	$a \to \{a \leftarrow a + 1, a \leftarrow a - 1\}; a \to \neg a$
SBR	Statement Block Removal	$stmt \to \emptyset$

Table 2.2: Mutant operators

Pseudo-Tested Methods

The operators mentioned in Table. 2.2.2 define mutations that mutate low-level parts of a program. But the concept of mutation testing can be extended to include other types of mutation operators. Niedermayr, Juergens, and Wagner [8] introduced a new set of mutation operators that mutate program code at a higher level. The concept of mutating a program at a higher level is often called extreme mutation testing. The idea is to define mutation operators that replace entire method bodies with a single return statement. As an example, for a method that returns a String, the method is mutated by removing all statements in the method body and inserting `return ""`, `return "A"`, or `return null`. If such mutants are not killed when executing the test suite, Niedermayr, Juergens, and Wagner [8] defines the method as being pseudo-tested.

2.2.3 Continuous Integration

Continuous Integration, or CI, is a practice of software development where multiple developers integrate their work frequently into a commonly shared master. Every change that is being integrated is being verified. The verification

can be done differently depending on the project. But the verification process commonly includes an automated build phase, automated testing phase and a code review. Larger projects usually use CI to reduce the risk of integrating bugs into the shared master. This reduced risk leads to more rapid development while writing new integrations [10].

2.3 Tooling

2.3.1 The CI Server

A CI server is a build management server solution. The CI server used for the project is a commercially available CI Server. The server lets the user define build configurations, or jobs, that define how a build should be executed in the CI job. Moreover, custom build agents can be connected to the CI server. The build agents can use any hardware configuration and operating system to suit the needs of the build configurations.

2.3.2 GitHub Checks

GitHub Checks is a feature that enables feedback from different CI tasks that are running after creating pull requests. GitHub Checks is provided as a separate tab in the pull request window on the GitHub pull request overview page. With Checks, it is possible to both annotate the updated code directly in the pull request, as well as produce a rich markdown report.

2.3.3 Gradle

Gradle is an open-source build automation tool with a focus on performance and flexibility [1]. Gradle uses build scripts that can be written using a Groovy or Kotlin DSL. Gradle is customizable in that it can be easily extended to fit the needs of a given project. It builds a dependency graph of all build targets and their dependencies on each other. This dependency graph gives information on build targets independent of each other. Gradle uses this information in two ways, first, it can execute independent targets in parallel on multiple threads. Moreover, Gradle caches compiled targets. Thus, if any of the independent targets is unchanged on a new compilation, Gradle can read the compiled target from the cache instead of recompiling it. Finally, Gradle is the official build tool for Android and is therefore widely used among Android developers.

[1] https://docs.gradle.org/current/userguide/userguide.html

2.3.4 Android Development Environment

Project Overview

We experimented on a large and complex Android project at a large music streaming company. There are on the order of a million lines of code in the project. At the time of experimenting, the project had release targets for ten different applications. Moreover, the project was heavily decoupled and modularized with more than a thousand subprojects contained under the root project. The reason for the decoupling was to enable developers to move fast and independently within the project. There were more than a hundred developers actively working on the project. Each of the developers worked in a team that had ownership of a certain set of subprojects.

Due to the decoupling, some of the subprojects did not rely on the Android API. This had implications on how unit tests were being executed on the subproject. The subprojects that had no dependency on the Android API used a JUnitTestRunner to execute the test suite. The subprojects with a dependency on the Android API used Robolectric [2] to mock the Android API and execute the test suite. However, to detect flakiness in Robolectric tests, a custom RobolectricTestRunner was used.

Continuous Integration for Android

The CI pipeline for the Android project was large. Developers merged up to a hundred pull requests every day. The average pull request change size, which is the number of insertions and deletions, was 70. Approximately 40 CI tasks were started on the CI server whenever a developer created a pull request. The collection of these CI tasks that are started on pull request creation is called the pre-merge task. The pre-merge task executes in a timely fashion to provide direct and immediate feedback to the developer in the pull request. This also implies that any new CI task must adhere to these time constraints. In Fig. 2.3 we display an overview of the average weekly build times for the pre-merge task on P25, P50, and P75.

[2] http://robolectric.org/

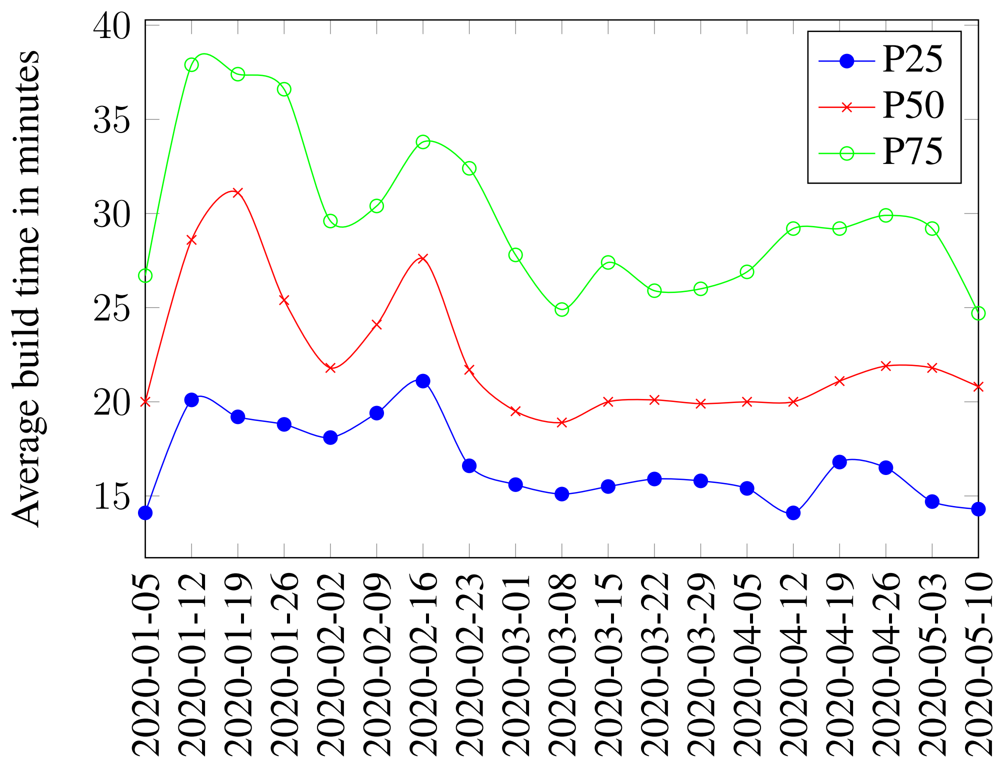

Figure 2.3: Build times for the Android pre-merge task week by week.

Chapter 3

Related work

In this chapter, we give an overview of work related to mutation testing and software quality in large-scale systems with high code and commit throughput.

3.1 Automated Fault Detection Tools in CI

In a paper by LaToza, Venolia, and DeLine [16], developers were asked to participate in surveys and interviews regarding their development behavior. In the research, the authors asked developers about their choice of development tools, activities, and practices, and their respective satisfaction with each. In the work, the author concluded that any tool developed to assist a developer must take into account the high demands of a developer's time.

Moreover, Layman, Williams, and Amant [17] explored developer behavior when using fault detection tools by surveying and interviewing developers. Based on survey results, the author outlined that these tools should be designed to properly fit into the development workflow and goals of the developer. Also, the tool must be precise and accurate in its descriptions of the faults. Finally, the tool must be reliable to build developer trust.

In another paper, Johnson et al. [18] investigated the use of static analysis tools during software development. They conducted interviews with developers and discovered that even though every developer thought these tools were useful, there existed barriers that made them hard to use. For example, the prevalence of false-positives and the way that errors are presented were common barriers for the developers. If a tool is presented as a standalone binary it tends to be used infrequently. The authors stressed the importance to implement these tools into the existing developer workflow to make them effective.

Sadowski et al. [19] attempted to solve the issues discussed above by in-

troducing a CI analysis framework for static analysis tools. The tool is called Tricorder and is used at Google for performing static analysis of their programs during code review. In Tricorder, if a tool discovers an error, it produces and surfaces a comment to the developer in the pull request. The pull request reviewer can then decide if the reported error should be fixed or if the reported error is not useful. The paper also discusses five key points in Google's philosophy on program analysis. Three out of five of these points are related to usability for the developer. First, there is a high priority on minimizing false-positives. Whenever a developer votes that a comment is not useful, a bug report is filed to the team developing the tool. If more than 10% of the comments from a tool are marked as not useful, the tool will be disabled from the code review process. Moreover, the tool should make data-driven usability improvements. Based on data from the votes, the authors found one example where 75% of the filed bug reports were due to confusing phrasing of the comment. Finally, workflow integration is key. It is important that the analysis process is triggered automatically by developer events such as editing code, running builds, checking-in changes, for example. If the analysis tool is a standalone binary, it cannot be expected that the tool will be run as frequently as intended.

As discussed above, Tricorder is used for multiple static analysis tools at Google. At Google, a single repository is used for most of the over two billion lines of code [20]. Due to this, the process of computing statement coverage is not straightforward. Ivanković et al. [12] explained how statement coverage is calculated at Google for more than one billion lines of code every day. Due to the scale of the number of changes, statement coverage can not be recomputed for the entire codebase on every change. Therefore, the authors proposed to only calculate the statement coverage of the changed code in a pull request using the Tricorder system. This facilitates the computation of statement coverage at scale, as well as the presentation of the coverage result to the developer during code review. Moreover, the authors performed a large-scale survey of developer interactions with the statement coverage system. Analyzing the results of the survey, the authors concluded that developers were in general positive about the idea of statement coverage. However, it was found that usability and low overhead of the tool are crucial factors for developers. This aligns with the findings in [18] and [17].

3.2 Mutation Testing

As discussed in Section 2.2.2, mutation testing is a methodology for evaluating the state of your program and it's test suite. There has been a lot of research into the topic of mutation testing and there exist multiple tools for running mutation testing in your language of choice. In a survey paper, Papadakis et al. [21] explores recent research into mutation testing. They discover that the most common mutation testing tool for Java is μJava [22], accounting for almost half of the research papers produced for Java. Moreover, Proteum [23] is the framework of choice for C, while Nutandis [24] is used for Javascript. Finally, the authors show that Java is the most popular language used in mutation testing research. It is used in more than half of the published papers. In this thesis, we will focus on mutation testing on Java. We do this for multiple reasons. Firstly, Android development is primarily done on the Java Virtual Machine (JVM) based languages. Secondly, the mutation frameworks for Java are in a mature state. Finally, since we are basing this research on previous research that uses Java mutation testing frameworks, it is natural to continue in that direction. In the following paragraphs, we will introduce some of the popular mutation testing frameworks for Java as well as a couple of surveys exploring the use of mutation testing in practice.

Mutation testing can be performed on a Java project in two ways. In the first case, the framework injects the mutant in the source code, recompiles the project, and then reruns the test suite. Representative tools for this approach is μJava [22], Major [25] and Jester [26]. The second method is to utilize the JVM and mutate the compiled bytecode. PITest [27], Jumble [28] and Javalanche [29] are examples of frameworks utilizing this approach. Of these frameworks, PITest is the only one that is actively maintained. PITest was designed to work with large projects and support different build systems such as Ant[1] and Maven[2]. It can be customized, both by supplying runtime arguments as well as by extending and overriding classes in the framework. For example, PITest can be customized to use another mutation engine, discussed in Section 3.3. Additionally, PITest performs extensive analysis to create a dependency graph of test cases and the line of code they exercise. With this information, PITest can be optimized to only run the dependent test cases when mutating a certain line, thus increasing performance.

Although PITest introduces an efficient way of executing mutation tests, the number of possible mutations increases rapidly as the program grows. In a

[1] https://ant.apache.org/
[2] https://maven.apache.org/

survey, Rossi [30] showed that the performance of mutation testing is impacted by (i) the statement coverage, and (ii) the project size. In the same survey, it was concluded that different mutation operators can be selected for faster and more relevant testing. However, it was found that the best mutation operators varied based on the specific needs of a project.

In another survey, Madeyski et al. [31] investigated the issue of the substantial runtimes for mutation testing, as well as the problem of equivalent mutants. One of the issues with mutation testing is that it can generate mutants that are equivalent to the non-mutated program. Listing 3.1 illustrates an equivalent mutant. The authors investigated different methods for identifying such equivalent mutants. One of the most effective solutions is to use higher-order mutations. That is, instead of introducing a single fault, two or more errors are introduced before each run of the test suite. Even though this approach is effective in mitigating equivalent mutants, the mutation testing loses some of its granularity.

```
1  public int eq(int b) {
2      int a = 2;
3      if (b == 2) {
4          b = b * a;
5      }
6      return a;
7  }
8
9  public int mut_eq(int b) {
10     int a = 2;
11     if (b == 2) {
12         b = b + a;
13     }
14     return a;
15 }
```

Listing 3.1: An example of a generated equivalent mutant

From the perspective of developers, there are mainly two problems with using mutation testing on their programs. The first one, as mentioned in section 2.2.2, is the substantial runtime. Another issue is that a mutation testing run is, with high certainty, going to lead to a large number of reported errors, which might overwhelm the developer. In a large codebase, it is quite certain that a lot of the errors will be located in parts of the code that the developer has never worked with. In a recent paper, Petrovic and Ivankovic [11] deployed Tricorder for mutation testing in the Google CI system. By integrating mutation testing in the CI, the mutation tests only ran on the changed code in the pull request. With this approach, the two problems related to runtime and non-related errors were mitigated. The number of possible mutations is

greatly reduced and the surfaced error will be on code recently touched by the developer. Moreover, the authors proposed a rule engine for selecting only a single mutation per line, thus reducing the runtime of the analysis. Since the system was deployed into the CI, the errors were reported to the developers during the code review process as an annotation on the mutated line of code. The person reviewing the pull request could then vote on the usefulness of the annotation.

Since mutation testing is a general principle, the mutation operators can be exchanged depending on what your problem is. Delgado-Pérez et al. [32] discuss a concept called performance mutation testing. The idea is new mutants are defined to evaluate and improve the detection power of performance tests. In the paper, the authors introduce seven new performance mutation operators by looking at real performance bug patterns. Moreover, they explore applying the concept at the source-code level on general-purpose languages. The authors discover that classical methods of mutation testing are not effective when assessing and improving performance tests. Instead, performance mutation testing is a promising methodology for these types of assessments.

3.3 Pseudo-Tested Methods

Niedermayr, Juergens, and Wagner [8] introduced the concept of pseudo-tested methods, as described in Section 2.2.2. This thesis will study the prevalence of pseudo-tested methods in an industrial context. We now give an introduction. The authors performed the extreme mutation analysis on 14 open source projects and concluded that the ratio of pseudo-tested methods varied heavily between programs, ranging between 6% to 53%. However, they also found that even well-tested programs with high statement coverage have pseudo-tested methods present. An advantage of their approach over regular mutation testing is a drastic reduction in runtime. A drawback is reduced granularity in discovered errors.

Building on the work by Niedermayr, Juergens, and Wagner [8], Luis Vera-Pérez et al. [33] developed a tool called Descartes that detects pseudo-tested methods. Descartes was built as a custom mutation engine for the PITest framework [27]. The authors performed an extensive survey comparing Descartes to the regular mutation engine in PITest by running both on 21 open source projects. Descartes provided a drastic improvement in runtime compared to the regular mutation engine used by PITest. However, naturally, the number of mutations caught by the test suite was higher when performing regular mutation testing. Finally, to assess the usefulness of the tool, the authors prepared

pull requests on eight projects, suggesting changes to mitigate the pseudo-tested methods. In all cases, their pull requests were merged into the projects, indicating that detecting and mitigating pseudo-tested methods is useful in improving software quality.

3.4 Mutation Testing on Android

Android projects have a different project and dependency structure compared to regular Java projects. For instance, Android projects have a substantial amount of code to define the user interface and dependencies in XML files during compilation. To address this, researchers have created multiple frameworks for running mutation tests on the Android platform.

Deng et al. [1] introduced a framework that extended the regular mutation operators listed in Table 2.2.2. The authors identified that even though Android applications have the highest number of releases and downloads among mobile applications, there is a prevalent quality problem. In their mutation testing framework, the authors introduced mutation operators for Android-specific functionality. This empowers the mutation testing to discover errors that platform-agnostic frameworks would not detect. For example, the framework introduces mutation operators that mutate event handlers and permission-related code. However, the authors concluded that the area required significant additional research. They did not consider all aspects of Android errors and did not have any comprehensive Android fault model.

In a more recent paper, Deng et al. [2] extended their previous study in [1]. The authors define new mutation operators to cover more of the different Android-specific functionalities. Moreover, they extend their fault model by performing additional empirical studies with real-world applications. Finally, the paper studies eight Android applications, instead of only one, as the first paper did[1].

Deng, Offutt, and Samudio [3] investigated the fault detection effectiveness of four existing Android testing techniques. Moreover, the authors collected occurring faults from nine different open-source Android applications. In addition, they also invited experienced Android developers online to hand-seed faults into the different applications. Based on this new method in gathering relevant faults, the authors derived six additional mutation operators compared to the operators derived in [1][2].

Jabbarvand and Malek [4] makes the observation that there are lots of companies investing heavily in mobile application development, but testing and verifying the applications is difficult. The authors built a complete tool for

running mutation testing on Android applications. The tool is called μDroid. Moreover, the authors derive ten types of Android problems that differ from previous research. The errors are related to Android energy consumption anti-patterns. Based on these errors, the authors introduce mutation operators to test the problems. The authors derived these by investigating pull requests and bug reports on Android repositories. Finally, the authors run the tool on 50 Android applications and conclude that their tool produces fewer mutants than general frameworks such as PITest. Though, the authors argue that their mutants are stronger compared to regular mutation testing mutants since they tie to Android-specific features.

Escobar-Velasquez et al. [5] identified a similar pattern. To get more effective mutation testing there is a need to define Android-specific mutation operators. They gave references to previous empirical studies showing that effective mutation testing requires mutation operators specified to the underlying domain. In this paper, the authors proposed two frameworks. The first framework is called MDroid+, which is also introduced fully in a separate paper[34]. MDroid+ is used to mutate open source applications. The second framework introduced is called MutAPK. MutAPK is capable of performing mutation tests on an Android APK package. They also systematically tried to create an Android fault model. They categorized more than 260 types of Android faults and grouped them into 14 categories. The underlying material used was gathered from more than 2000 artifacts publicly available such as bug reports and commits. Based on this defined taxonomy of common errors, the authors defined a set of 38 mutation operators. Finally, they introduced an infrastructure to seed these mutation operators into a real Android application.

Luna and Ariss [6] introduced another framework for performing mutation tests on Android. The tool is called Edroid and was designed by the authors to be extensible and flexible to provide value to future research. In the paper, the authors also defined ten new mutation operators for Android applications. The authors focused on mutating Android applications through the UI and configuration code, rather than mutating only the Java code. The authors show that Edroid was able to generate faults that were effective in revealing errors in the applications test suite.

Paiva et al. [7] performed similar work by defining customized mutation operators. As in the paper by Luna and Ariss [6], the authors focused on UI related problems. They defined new mutation operators to, for example, mutate specific parts of the Android lifecycle methods. They thereafter evaluated the operators with a tool called *iMPAcT*. With the help of *iMPAcT*, they evaluated their framework on 56 Android applications available on the Google Play

Store. Based on the study, they also discover improvements that can be made to the *iMPAcT* tool.

Another area of research for mutation testing on Android was conducted by Koroglu and Sen [35]. Instead of mutating the source code of an Android application, the authors proposed a methodology for mutating the test cases of an Android application to detect crashes. The idea is that the framework mutates a test case, runs the test suite, and if the test suite detects a crash, a fault has been detected. The authors mutate existing test cases or add additional test cases to detect crashes. For example, they add a mutation operator that examines unhandled exceptions or network-based problems. The authors were able to show that the approach was able to detect crashes in Android applications that had not been detected before.

Defining new taxonomies of common errors for each specific domain under test is effort-intensive. However, as discussed above, the domain-specific mutation operators are stronger than the general operators for that specific problem. Tufano et al. [36] introduce a new methodology for generating a new set of mutation operators for a given domain. To do this, they use a Neural Machine Translation approach that learns mutant operators from a large corpus of bug fixes. They define a complete architecture of their system where they outline each of the steps for generating the new mutants, as well as executing them on the program under test. The research is published as a framework called *DeepMutation*.

The introduction of additional mutation operators to increase relevance in the Android development domain is interesting. In this thesis, we are taking a similar approach to measuring the relevance and effectiveness of a new set of mutation operators for the Android domain. However, our approach differs since we take a step back and do not create mutation operators that are customized to the Android domain.

Chapter 4

Methodology

4.1 Experiments in CI

4.1.1 Technical Infrastructure

a. Overview

In the following sections, we explain our entire technical infrastructure that is made of multiple components. As shown in Fig. 4.1, the process starts when a developer creates a pull request. When the pull request is created, GitHub triggers the pre-merge task to start on the CI server. The bash script, introduced in Section 4.1.1.b, is created as a dependency to the pre-merge task and automatically starts executing when the pre-merge task starts. The bash script starts multiple custom Gradle tasks, explained in Section 4.1.1.c. As soon as the pre-merge task finishes, the developer gets notified via email. If we discover a pseudo-tested method in any of the methods methods in the pull request, we create a report under the GitHub Checks tab as shown in Fig. 4.3. When the developer gives feedback on any of the methods in the report, a request is sent to the feedback collection service, introduced in Section 4.1.3.d. The feedback collection service stores the feedback in a database for further analysis.

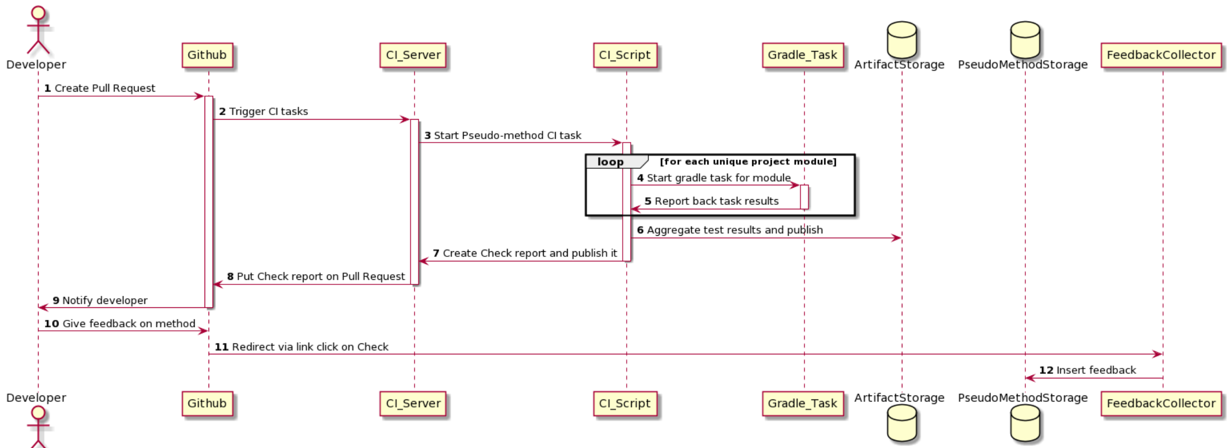

Figure 4.1: Architecture of Build Pipeline

Finally, after the collection of developer feedback, we as the build owners, trigger a local program that is responsible for an ETL procedure that loads artifact data into our database, as illustrated in Figure 4.2. Examples of artifacts include unmutated bodies of methods that were found to be pseudo-tested, as well as the method bodies of test cases exercising them. These artifacts are supplied by the Gradle task to the artifact storage, which is a database accessible over HTTP. This process is explained in further detail in Section 4.1.1e. Moreover, we create a tool that produces a list of developers that have a report on their pull request but have not provided feedback yet. We use this list to manually notify the developers and request feedback. We describe this further in Section 4.1.1e.

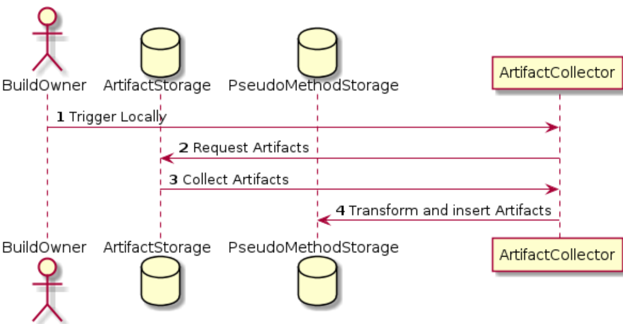

Figure 4.2: Flow of Retrieving and Storing Artifacts

b. Bash Script

When a developer creates a pull request, code modifications in the pull request usually span across multiple subprojects. Since we can only run our custom Gradle task on a single subproject at a time, we need a parent script that orchestrates the multiple Gradle tasks. We therefore use a bash script that has three responsibilities. First, the script performs a git diff against the base SHA of the pull request to find all of the updated Java files and the changed lines in each file. Next, the script finds the subproject for each Java file and triggers the Gradle task for each unique subproject. Finally, the script aggregates the outputs from the Gradle tasks to create a GitHub Check report. The script also aggregates raw data such as test cases and method bodies from the Gradle tasks and publishes them as artifacts that can be downloaded at a later stage. We constrain the bash script to timeout after 12 minutes to comply with the runtime constraints shown in Fig. 2.3.

c. Gradle Task

The project is built using Gradle. Therefore, to run our tooling on the project, we created a custom Gradle task. The Gradle task depends on PITest [27] and Descartes [9]. Both PITest and Descartes are built to be integrated into a Maven project. We make use of a Gradle port for PITest to make it compatible with the Android Gradle project under test [1].

The Gradle task we define has multiple responsibilities. We use command-line arguments to provide the task with a list of all the changed Java files and the changed lines in a pull request. The task then retrieves the classes that the pull request modifies. To identify the methods to mutate, we use a framework called Javaparser [37]. For each changed Java source file, we load the source file into Javaparser to create an abstract syntax tree of the source file. Thereafter, we look at each line that changed in the current source file. For each of these lines, we look at the methods that are located within the lines and save them. When the process is complete, we have a list of all the methods that the developer modified in the pull request.

PITest only accepts a command-line argument for *excludedMethods*, i.e., methods that must not be mutated. Therefore, we invert the list of modified methods to obtain the list of methods that were not modified in the pull request. Next, we provide the classes and the list of methods to exclude as arguments to PITest, which performs the mutation analysis using Descartes as the mutation engine. Finally, when the mutation analysis is done, the created report is

[1] https://github.com/koral–/gradle-pitest-plugin

translated into a standardized format that is published as an artifact by the CI script. If a surviving mutant is discovered, a markdown text is produced for the identified pseudo-tested method.

d. Feedback Collection Service

In the report that we publish on GitHub Checks, we include feedback buttons for each pseudo-tested method. We use an HTTP web server to record clicks from the developer on the feedback buttons. The links that represent the feedback buttons are URLs specifically crafted in the build step to be unique for the given reaction and method. When the developer visits the URL, we record it as a feedback entry in a database. Furthermore, to ensure idempotence, we always check if a feedback entry for a given method already exists. Finally, we allow the developer to give an additional comment where they, in free text, can elaborate on how they perceive the usefulness of the report.

e. Artifact Collection

During the pseudo-mutation testing phase on the CI server, we generate the report to be displayed as a GitHub Check, see Section 2.3.2. However, due to security reasons, the agent that runs the CI task runs in a sandboxed environment. It is not allowed to connect to services that are not whitelisted. The build script thus utilizes a feature of the CI task which is artifact publishing. Whenever a CI task completes, it can choose to upload the artifacts that it produces. Therefore, the Gradle task uploads aggregated data from the task as artifacts.

We develop a utility tool to retrieve all the artifacts, transform them, and record them in a database. We trigger the utility tool manually. The database is designed to make it easy to run arbitrary analytics on the data. In Table 4.1, we list the fields used to store information about the developer feedback.

Column	**Description**
method_id	The id of the method
head_sha	The SHA of the commit triggering the CI Script
reaction	The feedback (enum type) given on the method
comment	The additional comment given on the method
reaction_timestamp	The timestamp for when feedback was given

Table 4.1: Table containing the elements of feedback for a given method

In Table 4.2, we store information regarding the job that was executed on the CI server together with environment variables for the execution.

Column	**Description**
method_id	The id of the method
pr_branch	The branch that the task ran on
ghe_project	The root project of the pull request
commit_timestamp	The timestamp of the commit triggering the task
classification	The pseudo-tested classification of the method
mutation_line_number	The line number of the method in the source file
name	The name of the method
in_class	The class of the method
package	The package of the method
project_module	The subproject in the root project
method_source_code	The original, unmutated method body
file_source_code	The complete source code of the file containing the method
ghe_url	The URL to the check on GitHub
relative_file_path	The relative file path from the root project directory

Table 4.2: Table containing general data about the method under test

In Table 4.3 we store information about what mutation operator was applied and the status of that mutation operator, stating if it is killed by the test suite or not.

Column	**Description**
id	id
mutator	The mutation operator
status	Status displaying if the mutant was killed
method_id	The id of the method that was mutated

Table 4.3: Table containing data about the mutation and its result

In Table 4.4 we store information regarding each test case that is exercising the mutated code. We store the actual code of the test case as well as context variables of the test case.

Column	Description
id	id
name	The name of the test case method
test_method_source	The body of the test method
relative_file_path	The relative file path of the test case from the root project
github_url	The URL to the test case on GitHub
file_source_code	The source code of the file containing the test case
method_id	The id of the method under test

Table 4.4: Table containing data about the test cases exercising a method under test

f. Robolectric

The Android project uses Robolectric to run unit tests that depend on the Android API [2]. Robolectric creates a sandbox for each test case to isolate it from other test cases. After Robolectric creates the sandbox, it reloads the compiled Java bytecode from the disk. It does this to mitigate bytecode manipulations that are common on Android. However, when Robolectric loads the bytecode of a class from the disk, it becomes incompatible with PITest. Note that PITest had previously injected the mutated bytecode into the class that was already loaded in the JVM. Robolectric overrides these mutations by PITest since it loads the file from the disk. To mitigate this, we created a patch for the PITest mutation framework. When PITest injects the mutation into the class loaded in the JVM, we also write the mutated class code to the disk. In our scenario, this is a safe operation, since the mutation testing runs in a sandboxed CI environment at all times. Therefore, we can never run into the risk of accidentally releasing the mutated code. We made this patch for the PITest engine publicly available. It can be found as a fork of the PITest repository on GitHub [3].

4.1.2 Pilot Study

We initiated our study by letting our CI task run for one week as a pilot study. During the pilot study, we monitored the architecture to ensure that our entire pipeline was functioning as intended. Moreover, we gathered feedback from five developers on some of the reports during the pilot. After the pilot was concluded, we removed all entries stored in our feedback database. The

[2] http://robolectric.org/
[3] https://github.com/Kvarnefalk/pitest

purged database made sure that we did not use data from our pilot study while compiling our final results. The pilot was deployed between 2020-04-15 until 2020-04-21.

4.1.3 Feedback Collection

In the Android project, there is a lot of information presented to the developer at each pull request. The report we publish in the GitHub Checks tab never has a neutral status and never creates an error on the build. To increase its visibility and maximize developer response ratio, we create a simple utility tool. The tool finds a list of authors of pull requests with at least one pseudo-tested method. We then directly notify the developer over Slack [4], the internal communications tool, and ask them to provide feedback on the pseudo-tested method. When analyzing the results, we use the contents of these discussions and the additional comments that the developer writes after giving feedback on the reported pseudo-tested method.

4.1.4 Filtering of Reports

During the pilot study, we discovered that the mutation testing framework sometimes produces false-positives when running tests that depend on Robolectric. We are not interested in getting feedback from developers in those cases. Therefore, before we notify the developer that they have a pseudo-tested method on their pull request, we manually validate the report by pulling the changes in the pull request onto our local machine. Thereafter, we manually edit the source files according to the information in the report. Finally, we run the entire test suite for the subproject. If any of the test cases fail locally, we report the method as a false-positive in our database. If all test cases pass and the method is correctly reported as pseudo-tested, we send a message to the developer over Slack, notifying them of the report and ask them to provide feedback.

4.1.5 Reporting Format

In Fig. 4.3, we show the report for a single method that is found to be pseudo-tested. For each pseudo-tested method in the report, we start by highlighting the name of the method, followed by the class that the method is located in, and a link that takes the developer to the changed method in the GitHub pull

[4] https://slack.com/intl/en-se/

request, as shown in Fig. 4.4. Then we list the mutants that are not killed by the test suite. We also include a list of all the test cases that exercise the highlighted method. Finally, the developer can choose to give a reaction to the reported method by pressing on one of the five links at the bottom of the report. Table 4.1 lists the different reactions and their meanings. When the developer clicks on any of these links, they are redirected to the feedback collection service introduced in Section 4.1.3. The developer can choose to leave an additional comment on the redirected page as shown in Fig. 4.5.

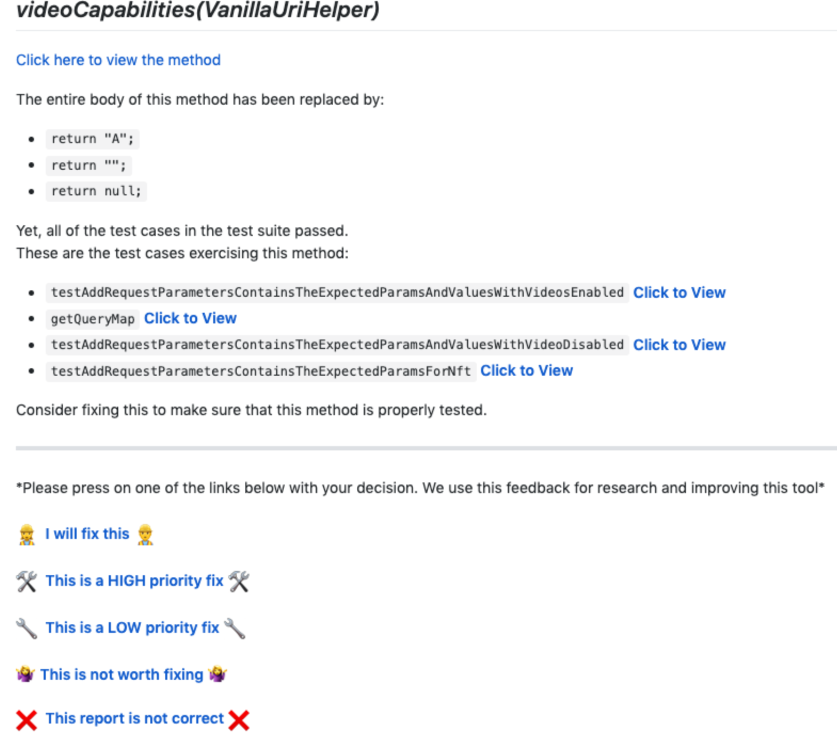

Figure 4.3: Example of the report for a single pseudo-tested method

```
177     private static String videoCapabilities(boolean videoCapable, boolean videoDrmCapable) {
178         final List<String> l = Lists.newArrayList();
179         if (videoCapable) {
180             l.add("video");
181         }
182         if (videoDrmCapable) {
183             l.add("video-drm");
184         }
185         return Joiner.on(',').join(l);
186     }
```

Figure 4.4: Example of a link to a mutated method in a GitHub pull request

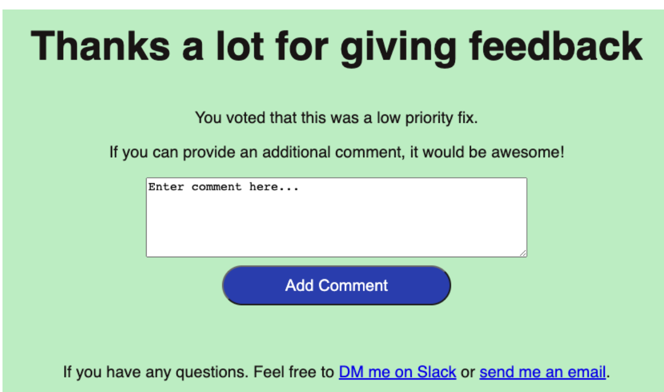

Figure 4.5: Example of the landing page after giving feedback on a pseudo-tested method.

Reaction	Description
I will fix this	The developer commits to fixing this
This is a high priority fix	The developer marks this as a high-priority fix
This is a low priority fix	The developer marks this as a low-priority fix
This is not worth fixing	The developer won't fix this
This report is not correct	Something is wrong with the presented report

Table 4.5: The list of possible feedback the developer could give for a pseudo-tested method

4.1.6 Experiment Duration

The experiment was deployed on 2020-04-22 and ran until 2020-05-28, which amounts to the duration of 24 working days.

4.2 Interviews

We conduct interviews with developers according to a methodology explained by Bengtsson [38]. The interviews aim to get an overview of how the developers interpret the report and to discover their opinions about the tooling and reporting format. We formulate five themes that the interviews explore. These are *Developer Background*, *Testing Weakness*, *Intent to Correct Weakness*, *Report Context*, *Tool Relevance Across Project*. We discuss the subject selection for our interviews in Section 4.2.1. Section 4.2.2 describes the formulation of our interview questions. Analysis of the conducted interviews is discussed in Section 4.2.3

4.2.1 Subject Selection

Before selecting the developers to interview, we examine our generated reports to get a list of all developers that have interacted with them. During and after the experimentation period, we send out invitations to these developers to participate in short interviews. We send a reminder if our first invitation is unanswered.

4.2.2 Response Collection

We conduct the interviews remotely over a video conferencing tool. At the start of each interview, we ask the interviewee for consent to record the interview for the purpose of this thesis. Upon approval, we record the interview and at a later stage, transcribe it manually. We conduct five interviews and the transcriptions are listed in Appendix A. During each interview, we share the GitHub Check report that the developer has previously interacted with. The set of questions are listed in Table 4.6. Furthermore, our interviews follow a semi-structured format. We base our interviews upon the question template but allow ourselves to ask follow-up questions to dig deeper into the answers and the opinions of the developer.

Q0	Briefly describe your work and role within the project
Q1	What are the testing weaknesses of the presented method?
Q2	Is this something that you have fixed or plan to fix?
Q3	What would be the most appropriate CI-status for this problem?
Q4	Is GitHub Checks suitable for reporting this?
Q4.1	Would it be better to present it using a website, for example, which allows for more interactivity in displaying the error?
Q5	Did you try to reproduce this locally?
Q6	Would you like to add anything to the report?
Q7	Would you like to remove anything from the report?
Q8	Are there any parts of the project you do not want to mutate?
Q9	Any other suggestions?

Table 4.6: List of questions asked during developer interviews.

4.2.3 Response Compilation

Since we base the interviews upon a standardized template, we can analyze all of the responses together. According to the methodology described by Bengtsson [38], we start by breaking the interviews up into meaning units. A meaning unit is the smallest unit that contains some of the insights the researcher needs. It is the constellation of sentences or paragraphs containing aspects related to each other, answering the question set out in the aim. This helps us extract key takeaways across different interviews. With the interviews de-contextualized into meaning units, we start categorizing the meaning units that share a similar meaning into the same theme. Finally, we summarize all of the key takeaways for each theme.

Chapter 5

Results

5.1 RQ1 - Usefulness of Presenting Pseudo-tested Methods in the CI

In Figure 5.1, we outline the number of responses from developers for each category during the course of the experiment. We separate the answers depending on if the pseudo-tested method is auto-generated or developer-written. We observe patterns that the feedback indicates that the developers do not believe mutation faults on auto-generated methods are as relevant as those for developer-written methods. To a large extent, developers do not think it is worth fixing pseudo-tested methods in auto-generated classes. However, developers seem more likely to intend to fix errors on methods that are written by a developer.

Even though additional comments are optional while providing feedback on a report. In Table 5.1, we show the comments that developers made while leaving feedback on auto-generated methods. Since the code is auto-generated, they never see the code that is being mutated. One of the comments highlights the fact that since the framework responsible for generating the code is made by a trusted third party, it does not need to be tested. It is enough to simply trust that it behaves as they would expect it to.

In table 5.2, we display the comments made by developers on developer-written methods. We can see that more developers have a positive sentiment towards the tooling in this case. However, where the methods are categorized as not being that useful, we receive comments indicating that code ownership, production Vs. debug builds, and refactoring changes can help to determine if a method is relevant to test.

Combining the feedback responses and the comments, we discover that

certain types of changes are categorized as more useful. Developers have a tendency of classifying developer written methods as more necessary to fix than auto-generated methods. Moreover, there are indications that developers believe refactoring changes and changes where the developer does not own the code is less useful. However, since the sample size is small, we are careful to only call it indications.

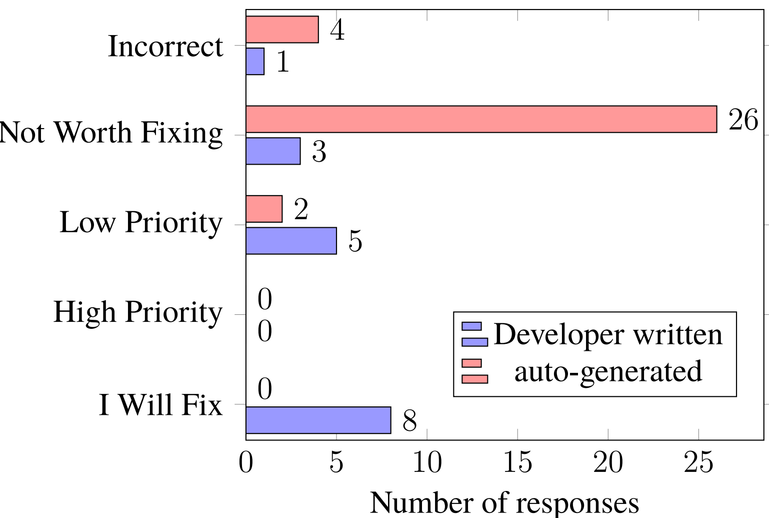

Figure 5.1: Feedback given for a total of 49 pseudo-tested methods out of 60 reported.

Developer	Reaction	Comment
Developer 1	Not worth fixing	"It is an auto-generated method that is out of the developers control"
Developer 2	Low priority	"All of our tests were testing expected equality and therefore relies on asserting equality. We should probably have negative tests asserting the model is not the same. But this is a low priority for us."
Developer 3	Not worth fixing	"The PR did not contain any changes related to the failing tests or method."
Developer 4	Incorrect	"It complains about auto-generated code which I am not testing here"
Developer 5	Incorrect	"This is an auto-generated class. The stated method in the report is generated. It is implemented by an external Google tool and I think we generally assume it to be correct. We never test the validity of these methods."
Developer 6	Not worth fixing	"A little confusing, especially considering it is about stuff not directly affected. Is is used somewhere deep in a dependent subproject."
Developer 7	Not worth fixing	"This will probably be less relevant as we move to Kotlin where there are data classes. Autovalue is an old Java trick to reduce boilerplate."
Developer 8	Not worth fixing	"My PR does not seem to touch the class of the method surfaced. Not sure why I am seeing this."

Table 5.1: Developer comments on auto-generated pseudo-tested methods

Developer	Reaction	Comment
Developer 1	Will fix	"I think it is a good catch. Even though the method is annotated with @NotNull the null value might happen at runtime."
Developer 2	Incorrect	"I did not change any implementation. I only renamed a field."
Developer 3	Will fix	"This is an excellent initiative. I can see this being very useful with single assertion testing where the number of tests can grow easily and you can miss things behind the false security of code coverage."
Developer 4	Not worth fixing	"I decided to not add a test to it since we did not have one before."
Developer 5	Not worth fixing	"This method is only used in debug builds and not in production. Thus there are no tests for it."
Developer 6	Low priority	"This code is used in production. But I was just cleaning up code and did not write this. I also don't own this part of the codebase."
Developer 7	Low priority	"This was affected by a set of test cases that I did not write myself. However, I believe that this is a really interesting feature and I would like to have it turned on for the project."
Developer 8	Will fix	"I will look at this when I have some time over."
Developer 9	Low priority	"That particular PR is updating a feature my particular team does not own. That is why I did not fix it."

Table 5.2: Developer comments on developer written pseudo-tested methods

5.2 RQ2: Developer Interviews

In total, we interview five developers. We contextualize and codify the interview responses into meaning units and connect them with the different interview themes. In Table 5.3, we highlight the process of how we distill meaning units from an interview, codify them, and assign them to the suitable interview theme. In the following sections, we outline the key takeaways for each interview theme. For the transcribed interviews, see Appendix A.

Meaning unit	Condensed Meaning Unit	Code	Theme
I have no idea about what that method is doing. Well, I can make a guess just from how they usually name stuff and so on in this team, but I don't know anything about this method per se. But in general, the app that uses this method doesn't have any UI, instead there are data methods that populate built-in UIs. So I'm thinking that this method is responsible for populating some data regarding the browse view or something.	I don't know what the method is doing. I can just assume based on naming conventions that these methods are responsible for providing data to another UI	Unsure of Weakness	Testing Weaknesses
Yes, absolutely it is enough that it is one false-positive, and people stop believing in it. I'm a victim of that myself.	You stop believing the tool on to many false-positives	false-positives	Tool Relevance Across Project
Maybe something like click here to generate a JIRA ticket or something that is prefilled with some information regarding this. Just to make it easy to put this as tech debt for the team. To put it as I would like to fix this but I can't do it now.	Generate pre-filled bug ticket in the report would be nice	Bug Ticket	Report Context

Table 5.3: Example of codification and thematization of interview responses

5.2.1 Background of Interview Subjects

All the developers we interview have interacted with the generated report and given feedback. During the interview, we ask them what their role is within the project. We discover that three of the developers work with and have responsibilities for specific features (subprojects) within the project. Two of the developers work cross-functionally in implementing functionality across the entire project, thus often working with features that they have no ownership of. The developers have different experience and seniority levels. For example, one of the developers is recently hired and one is the senior engineer of the team. One of the developers has worked a lot with testing practices and test certifications of programs.

5.2.2 Testing Weakness

We can see a pattern across our interviews suggesting that even though all the developers have interacted with the report, none of them could point out the root cause of why the tool discovers a pseudo-tested method. Although, one of the developers has an interesting observation. Methods in the project are sometimes designed to not do anything when they fail. When a method does not fail, it broadcasts an event to be received by event receivers. The developer figured that the pseudo-tested method can be related to that. It can be hard to properly assert that an event is not triggered from a method. The developer who is working cross-functionally received a report on a method that they never have seen during development (they implicitly touched it by renaming a field). Therefore, they have no ownership of the code under change and no idea why the tool reports the method.

Moreover, for two of the developers, the tool reports on auto-generated methods. It is clear that these methods produce more confusion than developer-written methods. The tool is mutating code that the developer never touched. Also, for the auto-generated methods, the tool reports the *equals* method to be pseudo-tested. However, some of the methods do not use the *equals* method directly. Instead, they resort to comparing attributes of the auto-generated object. This results in confusion from a potential false-positive and in understanding why the tool reports an error.

Finally, four out of five developers have a clear positive sentiment towards the tool. One believes that it discovers a potential error in the actual code, not the test suite. Others are positive that this type of tooling can drive the code and testing quality in the project forward.

5.2.3 Intent to Correct Weakness

Three out of five developers classify the error produced by the tool to be worth fixing. Although none of the three developers plan to fix the error right away. The developers who classify their methods as not worth fixing received reports on auto-generated methods. Two of the developers classifying their methods as worth fixing mention that it is hard to prioritize fixing tech debt while on tight deadlines. They are working on building new features with rapidly evolving functionality. For example, they mention that it can be hard to refactor a test class when your product manager wants you to release rapidly.

Another interesting note taken from two developers is the notion of bug ticket creation. One of the developers who blame tight deadlines as a factor for not fixing the bug, created a bug ticket and placed it in their team's TODO list. Another developer, who did not own the code under change, proposes a functionality in the report to let the developer automatically create a bug ticket pre-filled with the necessary information that gets assigned to the team owning the code.

Even though at least three of the developers plan to fix the error in the future, only one of the developers explored the pseudo-tested method by reproducing it locally. The other developers believe the information in the report is sufficient.

5.2.4 Report Context

Based on the interviews, we branch this theme into two main categories, the CI-status of the report and the actual content of the report on GitHub. We discuss these two categories in the following two sections.

CI-Status

As mentioned in the methodology, our report has a neutral CI-status, thus never failing the pre-merge build. All of the developers have a clear opinion that this type of tooling should not be neutral. There is a clear consensus on having the CI-status as one that needs developer acknowledgment. Developers still want to potentially exclude some methods where they know better than the tool. One of the developers states that when you have to merge a critical fix into master, it is not advisable to fixate on technical debt. It is also highlighted multiple times that a premise for moving towards a more aggressive CI-status is to increase the robustness and reliability of the tool. Otherwise, the developers on the project may get frustrated.

Content of the Report

During the interviews, we specifically ask if GitHub Checks is a suitable place to report these types of errors. In general, the developers appreciate the format of the report and the fact that it places itself on the pull request. However, it is evident that there is a significant cognitive load by the number of checks created on a pull request. As a default, ten checks execute with a neutral CI-status on each pull request. One of the developers states that they never look at the checks. If the pull request is green, they consider it as good to go. However, two of the developers mention that they discovered the tooling via the GitHub checks code annotations. Both of the developers enjoy that feature and believe it is nice to have the check interlaced with the code changes. According to them, it is much better to have it with the changed code. However, they want the annotation to provide more value by containing more information.

In general, the developers seem quite happy with the structure of the report, though, there are interesting opinions worth mentioning. To one of the developers, it was unclear before the interview if the tool was mutating test classes or the actual class under test. Another developer, with a pseudo-tested method having a large number of exercising test cases, had an interesting note. Since the sheer number of test cases displayed can be overwhelming, the developer suggested that we could potentially use some heuristic to highlight the test cases most dependent on the actual code. The same developer also proposed running A/B testing on the report and evaluate developer reactions. Two of the developers wanted a bit more clarification on how our tool worked, by potentially providing a step-by-step process and interlacing code interactively in the report. Another developer proposed color-coding the report, marking the most important parts with red.

5.2.5 Tool Relevance Across Project

From the interviews, we attempt to analyze what developers perceive of the usefulness of applying the tool across the project. One of the developers did not have any opinion on this topic. Three out of the remaining four developers expressed strong opinions that the tool should run on all parts of the code. It does not matter, according to the developers, if the code is auto-generated, part of a debug build, includes private methods, or if the developer making the change does not own the code. One developer is opposed to the idea of running the tool on auto-generated code. The reasoning is that the developer's team never writes unit tests for auto-generated methods since they assume that the frameworks work and already are properly tested.

One of the developers discussed elevating the tooling to create an overall metric for the entire project on the state of pseudo-tested methods. The developer expressed concern that information regarding the state of the project would be siloed in the pull request domain.

5.2.6 Key Takeaways

A clear takeaway from the interviews is that the developer's perception of the tool changes while discussing the tool and problem in depth. Education, which increases understanding about the tooling, seems to lead to an improved perception of its usefulness. Our report can be improved to navigate the developer through the problem. Moreover, even though results in Figure 5.1 indicate that auto-generated methods are not as useful as those written by developers, during the discussions in the interviews developers expressed that auto-generated methods should still be evaluated by the tooling.

5.3 RQ3: Developer Actions

According to Figure 5.1, there are eight methods marked as *I will fix*. Fifteen working days after the experimentation period finished, we manually ran PItest with the Descartes mutation engine on methods that was marked as *I will fix*. We discover that even though the developers do say they were going to fix the methods, they did not follow through. We also looked at if the developer created any pull request or made changes to their local branch to mitigate the error. We could not see any indications that developers were taking actions on fixing the methods.

We also observed that developers within the project felt that they were pressed for time during the experimentation period. During one of the interviews, a developer witnessed that they had a lot to do and would not have time to fix the issue in this sprint. The developer hoped to get it fixed by the next sprint but felt uncertain if that would be possible. Another developer also witnessed about being stressed for time and having tight deadlines. They noted that it felt hard to work on technical debt while you have a lot of new features that you need to implement. Moreover, it was clear while sending out meeting invites that a lot of the developers were busy. The interview invitations had a quite low response ratio. Multiple developers blamed busy schedules and tight deadlines as the reason for declining the interview.

Since the response *I will fix* is not tied to any bug ticket or backlog, it is hard in following up soft work on how the issue is handled. For example, we

cannot capture events such as planning to fix a mitigation or adding it to the current sprint. One developer mentioned during an interview that he created a bug ticket and placed it in the backlog of his team. However, the ticket was still located in the backlog three weeks after he added it to the backlog.

5.4 RQ4: Classification of Pseudo-tested Methods

In Figure 5.2 we show the ratio of methods reported as false-positives to correctly reported methods. In total, 188 methods are reported.

Moreover, in Figure 5.3 we display the distribution of method classifications across the entire experimentation period. As we can see, the majority of the methods do not have any test coverage. Approximately one-third of the methods are tested correctly, while only about 1 method per 100 methods is classified as being pseudo-tested.

We note that the large majority of the methods tested are not classified as pseudo-tested. Moreover, of the methods classified as pseudo-tested, more than 70% are categorized as a false-positive.

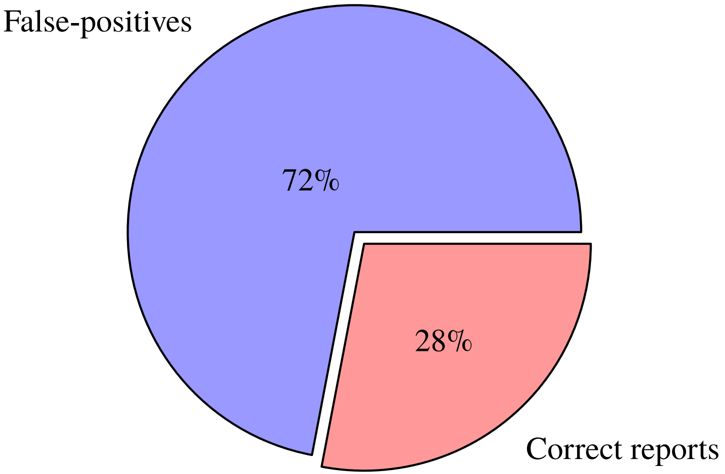

Figure 5.2: Distribution of false-positive and correct pseudo-tested methods

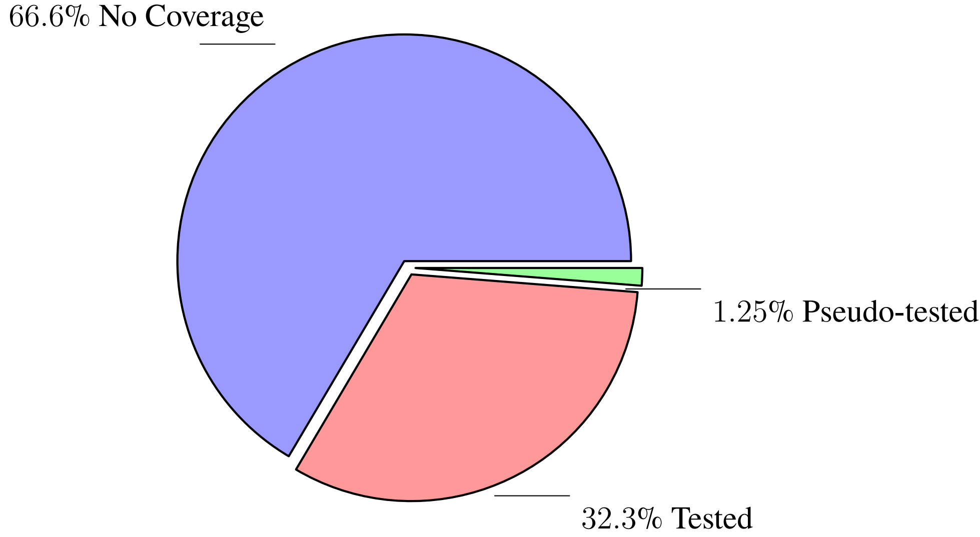

Figure 5.3: Classification distribution of examined methods

Chapter 6

Discussion

This chapter discusses five factors that allow us to draw conclusions about our research questions. We first outline threats that can impact the validity of this study, such as the robustness of the tooling, or the limited amount of data. Second, we discuss auto-generated methods. Third, we discuss how faults can be presented to developers to increase the relevance of the tooling. Then we go through why developers are not correcting faults they say they will fix. Finally, we reason about when the tooling is relevant to use and when it is not.

6.1 Threats to Validity

6.1.1 Robustness of the Tooling

There are a significant number of false-positives generated by the tool, as shown in Figure 5.2. The root cause of the false-positives is still not known. We suspect that it might be related to how PITest interacts with the Robolectric framework, as discussed in Section 4.1.1 f. Our mitigation for the problem seems to correct some of the issues, but there is room for further improvement.

In general, unit testing on Android is complex. Multiple libraries related to Android-specific functionalities need to be mocked. To mock a library, Robolectric instruments it by changing the bytecode in the JVM. Thus, even before PITest adds its own instrumentation, there is a lot of pre-existing instrumentation in the JVM. This makes the process of finding the root cause of the error complicated. It should be noted that the patch we apply to Robolectric is a potential source of the error. The Robolectric and PITest frameworks are complex. For example, if Robolectric is using internal caching for some of its test cases to optimize runtime, we would fail, since it would not pick up the

mutation that we write to the disk. We discussed the error with the maintainer of the PITest project[1]. He mentioned that they have seen similar errors of Java libraries depending on the Javassist framework [39]. However, he described that the workaround at PITest is to rewrite the bytecode of the class in Javassist doing the loading from disk. They mutate the bytecode in Javassist to load the mutated code instead. Since the errors are similar, we could potentially utilize the same approach. In that case, we would instrument the compiled Robolectric framework and change the way it loads classes. Unfortunately, this requires a deep understanding of both PITest and Robolectric, something we do not have time to acquire properly during the scope of this thesis.

In terms of this thesis, it is unfortunate that we have a significant ratio of false-positives over correct errors. Previous research in the related works section highlights the importance of robust tools. Multiple developers mentioned this during the interview as well. Since the CI-status of our report is neutral, the developer does not have to look at the report if they do not want to. If the tool is producing a false-positive, the developer has a high tendency of not looking at the tool again. Multiple interview subjects emphasized that sentiment. However, we are careful in manually going through each of our data points to ensure correctness in the reported data. Thus, none of the reported results contain false-positive methods. Although we should remember that if a developer sees a false-positive method before they see a correctly reported method, they can be biased against the tooling when giving feedback.

The current presence of false-positives leaves a great opportunity for future work. Both from a technical standpoint on how to develop a mutation testing framework for Android that does not produce false-positives, but also to investigate how developer sentiment potentially changes towards the tooling when it is more robust.

6.1.2 Limited Scope of the Study

Even though the experiment was ongoing for a total of 24 working days, only 60 methods were reported to developers. Out of the reported methods, 50 methods received feedback. The reason is both due to the high number of false-positives and the small amount of pseudo-tested methods across the project, see Figure 5.3. Due to the small amount of data, it is hard to do any reasoning about the statistical significance of the feedback. Instead, we have to resort to combining the given feedback with the interviews. Even though we apply a rigorous and methodological process in analyzing the interviews, there is a

[1] https://github.com/hcoles/pitest/issues/744

risk of underlying bias in the analysis process.

6.2 Auto-generated Methods

Based on the results shown in Figure 5.1 and the developer interviews, it is evident that the case of auto-generated methods is special. Also, there is a slight mismatch in the developers' perceptions of auto-generated methods between the interviews and the automated feedback gathering. When gathering the feedback, developers do not think that the issue is worth fixing. However, during the interviews, only one of the developers oppose testing auto-generated methods. Three of the other developers think it is still relevant to do.

All methods in our report are presented in the same format, irrespective of whether they are auto-generated or developer-written. Therefore, for auto-generated methods, multiple developers comment that they are not responsible for the method being mutated. Even though when we inspect their change, we can see that they change the template class to be generated. First of all, it seems to be non-intuitive to present auto-generated code to the developer in a report without giving further context. After all, the developers have never seen the code under mutation, since it is picked up by PITest in the JVM, after compilation. We believe that if auto-generated methods should be a part of tools like these, an effort should be made in explaining the steps carefully in the report. This aligns with the findings in our results. The initial feedback in the automatic response gathering is that the errors are not worth fixing. However, after discussing the tooling further, almost all of the developers believe that these types of errors are relevant. This indicates that insight about the tooling and clear steps on what the tool is doing increases the relevance perception of the tooling.

Furthermore, developers are opinionated about the concept of testing auto-generated methods. As we can see in Table 5.1, Developer 5 is of the opinion that these frameworks can be generally assumed to be correct. This sentiment is shared with one of the developers during an interview, see Section 5.2.5. On the other hand, three other developers during the interviews have opinions to the contrary. Due to the contradictory nature of this aspect of our tool, there are multiple ways forward. Either, the project or the organization takes a clear stance on how third-party libraries should be tested. This would help align the mismatch in opinions we observe across the project. Otherwise, if the different teams define their own stance and opinions, it could make sense to extend the tooling. For example, the tool can be extended to include a flag for not mutating auto-generated code. It would then be optional for teams if

they would like to opt-in.

6.3 Fault Presentation

During the developer interviews, we ask multiple questions relating to how the developers want faults to be presented on the pull request, both in terms of the actual structure and formatting of the report as well as the CI-status of it.

6.3.1 Structure of the Report

In general, developers appreciate the structure of the report. They also approve of its integration into GitHub Checks. Also, two of the developers discovered the tool via the code annotations possible via GitHub Checks. Even though the two developers only support plain text, they think it is great to have tooling interlaced with the code changes. This sentiment should be valuable for GitHub in continuing the development of features to support richer code annotations. With richer annotations, we could display what mutation is applied directly on top of the changed code which could lead to less confusion.

As discussed in Section 6.2, the report should clarify the process of mutating auto-generated methods since that is a special case. Moreover, some of the developers want more visualization of the mutations that we apply, potentially by including the unmutated and mutated source code side-by-side in the report. Another interesting idea by a developer is to display test cases better. For example, instead of listing the test cases, we could group them by dependency distance to the tested method. This would be helpful for developers since they would be able to debug the fault faster. If a developer would need to make a fix, it would also be easier to know which test case would be appropriate to fix. The same developer proposes adding color-coding to the report to quickly highlight the most important parts of the code. There are a lot of different aspects of the report that we can tweak and improve. Since the interviews highlight that a better understanding of the tool seems to increase the perceived usefulness of the different methods, this is an important area to continue exploring. Future research could make use of thorough user experience research and A/B testing to determine the most effective reporting format.

Many of the developers are introduced to the concept of pseudo-tested methods for the first time while seeing the report. This suggests that we could make a stronger effort in clearly outlining why and what the tool is doing in the report. An alternative, or even an addition, to having the information within the report could be to have an educational session with the developers when

the tool is launched. In such a forum, relevant opinions and questions could be discussed to increase the understanding of the tooling.

6.3.2 Bug Ticket

During the interviews, two of the developers mention bug tickets as a way of logging the faults that the tool detects. One of the developers created a bug ticket once they saw the report. Another developer proposes that there should be an option in the report for generating a bug ticket with the relevant information filled out. That is an interesting idea that provides value in multiple ways. First of all, we introduce the option to automatically create an action item for the developer, which makes it easier for them not to forget the fault. Moreover, bug tickets would provide a great benefit to us as tool developers since the reports with a bug ticket can be categorized as more relevant than reports with no bug ticket. Thus, bug tickets would be a great addition to the report and something that would be interesting to explore further in future work.

6.3.3 Status of Checks

A clear concern, highlighted by multiple developers, is that there are a lot of different checks being run when creating a pull request. First of all, there are over 40 build configurations on the CI server that can potentially fail. Moreover, there are at least 10 GitHub checks presented on each pull request, all with a neutral CI-status. This places a large cognitive load on the developers. It is also effort-intensive to dig through all of the neutral checks to discover a potential error. In the case of our tooling, as we can see in Figure 5.3, only about 1% of the tested methods are reported as pseudo-tested. Thus, for a large number of pull requests, the developer looks at the Checks tab just to find no errors reported.

Moreover, every developer is in favor of having a CI-status that needs their acknowledgement. The stated reasons align with the above discussion. One of the developers states that they never look at neutral reports. Another says that they stop looking at a report if it is not useful. Therefore, all of the developers are positive towards getting presented with an error that stops the build. However, all of them want the option to override the tooling, as would be possible in an acknowledgment-based approach. The preferred way of overriding a build-failure differs between developers. One of them wants to place annotations in the code while another wants to have a button within the report on the pull request. The question of evaluating the appropriate CI-status serves

as an interesting extension to this work.

Finally, it should be noted that the acknowledgment-based approach is conditioned on a robust tool. If a tool should break pull requests, it is essential that it is not producing false-positives. This opinion is highlighted by several of the developers. However, for these types of tools, it is also essential to get reports on false-positives. With a neutral report, developers tend to ignore instead of reporting. We received no complaints of false-positives from developers seeing the report, without us explicitly asking. Therefore, to aid the development of tools like this, it is imperative to have good developer feedback. Thus, there is a trade-off between a tool being neutral and bad, and being build-breaking and bad but at least get a direction on where to move to make the tool better.

6.4 Correcting a Fault

From the results in Section 5.3, we see that none of the developers that committed to fixing their faults actually fixed their faults during the scope of this study. There can be multiple reasons for this. First of all, there are only eight methods marked by developers as, *I will fix*. That is a small sample to draw clear conclusions from. Moreover, when conducting interviews with the developers, we discover interesting reasons for why the errors are not fixed right away. Two of the developers mention, as discussed in Section 5.2.3, that they want to fix the errors but have a lot of deadlines, which makes it hard to prioritize fixing tech debt like this. It is quite clear from the report that the tool is part of a research project. That can be another reason for the developers not feeling urged to correct the fault in a timely manner.

6.5 When is a Method Change Relevant?

It is hard to identify features that determine if a developer perceives a pseudo-tested fault to be useful or not. Since we do not have a lot of collected responses, we cannot draw clear conclusions based on the responses alone. However, while discussing the theme during the interviews, there is a clear opinion towards running the tooling on all code changes. However, reading through the additional comments in Table 5.2, we can see that refactoring changes are causing negative sentiment. A typical example is a developer renaming a field, such as in the case of Developer 2 in Table 5.2. If the field is present in multiple methods, all of these methods are considered to be changed and are therefore under test, even though the semantics of the methods remained

unchanged. This can potentially lead to the developer seeing pseudo-tested faults on methods the developer never has seen or interacted with.

Instead of restricting the tool to not run on certain parts of the code, it seems like the algorithm for detecting changed methods could be improved. For example, one could argue that a method that has only been refactored and is equivalent semantically, is not relevant to test. Even though it makes sense to test it, there is potentially a higher risk of the developer not knowing the implementation details of the changed method during a refactoring. However, implementing such an algorithm is not necessarily trivial. Therefore, this aspect of developing a more sophisticated algorithm to determine which methods to test leaves interesting opportunities for future work.

Chapter 7

Conclusions

To improve the quality of programs, it is important that they are thoroughly tested. Mutation testing in general, and extreme mutation testing in particular, are efficient ways to increase program testing quality. With the rising complexity of mobile applications, quality issues become even more important to consider. Detecting pseudo-tested methods in the CI system is an interesting approach where mutation testing is integrated into the natural development lifecycle without causing the long runtimes or delays often associated with mutation testing. In our study, we address the usefulness of utilizing pseudo-tested methods in this environment, and the possibility of classifying these methods as relevant. We also attempt to understand if developers intend to take action on the presented faults.

7.1 Usefulness of Pseudo-tested Methods

There are indications that pseudo-tested methods provide a useful means of fault-detection for developers. Developers were positive in general towards having this type of tooling in the CI system. However, it became evident that certain types of methods, e.g. auto-generated methods, were perceived to be less useful compared to methods written by developers.

7.2 Developer Action on Reported Faults

We discovered that even though some developers committed to correct the faults, none of them followed through. However, our sample size was small, which makes it harder to draw a clear conclusion.

7.3 Relevance of Presented Methods

There are certain factors that can determine the usefulness of a method presented in our report. First of all, large refactoring changes tend to produce more irrelevant pseudo-tested methods in the report due to the developer not actually changing any implementation details of the reported pseudo-tested methods. Also, auto-generated methods were perceived as less useful, although that sentiment varied across the project. In general, the sentiment seems to be that all methods are relevant for mutation. However, the algorithm that determines if a method has been changed could be improved to discover if a method was changed semantically. Based on comments on reported methods, that would be beneficial to improve the usefulness of our tool and relevance of the highlighted methods.

7.4 Future Work

The main limiting factor in this work is the prevalence of false-positives. One clear direction for future work is to determine the root cause of the false-positives. If this error is mitigated, a more rigorous study can be conducted to retrieve even more data. We also suggest additional research into the aspect of structuring the report to determine how it can be better formatted to improve the relevance of the tooling. Moreover, we believe that additional research can be made into determining what methods should be mutated in every pull request, for example, by introducing a more sophisticated diff algorithm.

Bibliography

[1] Lin Deng et al. "Towards mutation analysis of Android apps". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2015 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., May 2015. ISBN: 9781479918850. DOI: 10.1109/ICSTW.2015.7107450.

[2] Lin Deng et al. "Mutation operators for testing Android apps". In: *Information and Software Technology* 81 (2017), pp. 154–168. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.04.012. URL: http://www.sciencedirect.com/science/article/pii/S0950584916300684.

[3] L Deng, J Offutt, and D Samudio. "Is Mutation Analysis Effective at Testing Android Apps?" In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. July 2017, pp. 86–93. DOI: 10.1109/QRS.2017.19.

[4] Reyhaneh Jabbarvand and Sam Malek. "MDroid: An Energy-Aware Mutation Testing Framework for Android". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 208–219. ISBN: 9781450351058. DOI: 10.1145/3106237.3106244. URL: https://doi.org/10.1145/3106237.3106244.

[5] Camilo Escobar-Velasquez et al. "Enabling Mutant Generation for Open- and Closed-Source Android Apps". In: *IEEE Transactions on Software Engineering* (Apr. 2020), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/tse.2020.2982638.

[6] E Luna and O E Ariss. "Edroid: A Mutation Tool for Android Apps". In: *2018 6th International Conference in Software Engineering Research*

and Innovation (CONISOFT). Oct. 2018, pp. 99–108. DOI: 10.1109/CONISOFT.2018.8645883.

[7] Ana C.R. Paiva et al. "Testing when mobile apps go to background and come back to foreground". In: *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2019*. Institute of Electrical and Electronics Engineers Inc., Apr. 2019, pp. 102–111. ISBN: 9781728108889. DOI: 10.1109/ICSTW.2019.00038.

[8] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. "Will My Tests Tell Me If I Break This Code?" In: *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*. CSED '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 23–29. ISBN: 9781450341578. DOI: 10.1145/2896941.2896944. URL: https://doi.org/10.1145/2896941.2896944.

[9] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry. "Descartes: A PITest Engine to Detect Pseudo-Tested Methods: Tool Demonstration". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 908–911. ISBN: 9781450359375. DOI: 10.1145/3238147.3240474. URL: https://doi.org/10.1145/3238147.3240474.

[10] Martin Fowler. *Continuous Integration*. URL: https://martinfowler.com/articles/continuousIntegration.html.

[11] G Petrovic and M Ivankovic. "State of Mutation Testing at Google". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. May 2018, pp. 163–171.

[12] Marko Ivanković et al. "Code Coverage at Google". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 955–963. ISBN: 9781450355728. DOI: 10.1145/3338906.3340459. URL: https://doi.org/10.1145/3338906.3340459.

[13] Frances E Allen. "Control Flow Analysis". In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: https://doi.org/10.1145/390013.808479.

[14] Richard A Demillo and A Jeeerson Ooutt. “Constraint-Based Automatic Test Data Generation”. In: *IEEE Transactions on Software Engineering* 179 (1991), pp. 900–910.

[15] James H. Andrews et al. “Using mutation analysis for assessing and comparing testing coverage criteria”. In: *IEEE Transactions on Software Engineering* 32.8 (Aug. 2006), pp. 608–624. ISSN: 00985589. DOI: 10.1109/TSE.2006.83.

[16] Thomas D LaToza, Gina Venolia, and Robert DeLine. “Maintaining Mental Models: A Study of Developer Work Habits”. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE ’06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 492–501. ISBN: 1595933751. DOI: 10.1145/1134285.1134355. URL: https://doi.org/10.1145/1134285.1134355.

[17] L Layman, L Williams, and R S Amant. “Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools”. In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. 2007, pp. 176–185. DOI: 10.1109/ESEM.2007.11.

[18] Brittany Johnson et al. “Why Don’t Software Developers Use Static Analysis Tools to Find Bugs?” In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. IEEE Press, 2013, pp. 672–681. ISBN: 9781467330763.

[19] C Sadowski et al. “Tricorder: Building a Program Analysis Ecosystem”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. May 2015, pp. 598–608. DOI: 10.1109/ICSE.2015.76.

[20] Rachel Potvin and Josh Levenberg. “Google’s monolithic repository provides a common source of truth for tens of thousands of developers around the world”. In: *COMMUNICATIONS OF THE ACM* 59.7 (2016). DOI: 10.1145/2854146. URL: http://www.bazel.io.

[21] Mike Papadakis et al. “Mutation Testing Advances: An Analysis and Survey”. In: *Advances in Computers* 112 (Jan. 2019), pp. 275–378. ISSN: 00652458. DOI: 10.1016/bs.adcom.2018.03.015.

[22] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. “MuJava: An Automated Class Mutation System: Research Articles”. In: *Softw. Test. Verif. Reliab.* 15.2 (June 2005), pp. 97–133. ISSN: 0960-0833.

[23] Márcio Eduardo Delamaro, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. "Proteum/IM 2.0: An integrated mutation testing environment". In: *Mutation testing for the new century*. Springer, 2001, pp. 91–101.

[24] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. "Efficient JavaScript mutation testing". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 74–83.

[25] René Just. "The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 433–436. ISBN: 9781450326452. DOI: 10.1145/2610384.2628053. URL: https://doi.org/10.1145/2610384.2628053.

[26] Ivan Moore. *Jester-a JUnit test tester*. Tech. rep.

[27] Henry Coles et al. "PIT: A Practical Mutation Testing Tool for Java (Demo)". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 449–452. ISBN: 9781450343909. DOI: 10.1145/2931037.2948707. URL: https://doi.org/10.1145/2931037.2948707.

[28] *Jumble*. URL: http://jumble.sourceforge.net/.

[29] David Schuler and Andreas Zeller. "Javalanche: Efficient Mutation Testing for Java". In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 297–298. ISBN: 9781605580012. DOI: 10.1145/1595696.1595750. URL: https://doi.org/10.1145/1595696.1595750.

[30] Bruno Rossi. "Is Mutation Testing Ready to Be Adopted Industry-Wide". In: *Lecture Notes in Computer Science* 10027.November (2016), pp. 198–214. DOI: 10.1007/978-3-319-49094-6. URL: http://link.springer.com/10.1007/978-3-319-49094-6.

[31] Lech Madeyski et al. "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation". In: *IEEE Transactions on Software Engineering* 40.1 (2014), pp. 23–42. ISSN: 00985589. DOI: 10.1109/TSE.2013.44.

[32] Pedro Delgado-Pérez et al. "Performance mutation testing". In: *Software Testing, Verification and Reliability* (Jan. 2020). ISSN: 0960-0833. DOI: 10.1002/stvr.1728. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1728.

[33] Oscar Luis Vera-Pérez et al. "A comprehensive study of pseudo-tested methods". In: *Empirical Software Engineering* 24 (2019), pp. 1195–1225. DOI: 10.1007/s10664-018-9653-2. URL: https://doi.org/10.1007/s10664-018-9653-2.

[34] Kevin Moran et al. "MDroid+: A Mutation Testing Framework for Android". In: (2018). DOI: 10.1145/3183440.3183492. URL: https://doi.org/10.1145/3183440.3183492.

[35] Yavuz Koroglu and Alper Sen. "TCM: Test Case Mutation to Improve Crash Detection in Android". In: *Fundamental Approaches to Software Engineering*. Ed. by Alessandra Russo and Andy Schürr. Cham: Springer International Publishing, 2018, pp. 264–280. ISBN: 978-3-319-89363-1.

[36] Michele Tufano et al. "DeepMutation: A Neural Mutation Tool". In: (), p. 2020. DOI: 10.1145/3377812.3382146. URL: https://doi.org/10.1145/3377812.3382146.

[37] Roya Hosseini and Peter Brusilovsky. "Javaparser: A fine-grain concept indexing tool for java problems". In: *CEUR Workshop Proceedings*. Vol. 1009. 2013, pp. 60–63.

[38] Mariette Bengtsson. "How to plan and perform a qualitative study using content analysis". In: *NursingPlus Open* 2 (2016), pp. 8–14. DOI: 10.1016/j.npls.2016.01.001. URL: http://dx.doi.org/10.1016/j.npls.2016.01.001.

[39] Shigeru Chiba. "Javassist—a reflection-based programming wizard for Java". In: *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*. Vol. 174. Citeseer. 1998, p. 21.

Appendix A

Transcribed Interviews

A.1 Interview 1

John: *Could you just very briefly describe your role and work within the Android project?*
Interviewee: *I am Android developer at a team at the company where I work with integrating voice functionality into the project.*
John: *Okay cool. The plan for this meeting is basically so. So you've seen the report sent to you over slack. That is one part of the thing I am doing in my thesis where i basically like show methods to developers and see how they feel about the presented methods. Then I plan to do some of these interviews where I basically ask some deeper questions about this concept, how it can be improved, what you think about it, how it is having it presented in CI/pull requests page and so on. The thing I would like to start with is to share the report that you've already looked at.*
Interaction: *Showing the report on screen*
John: *Why is this method pseudo-tested or why does it show an error here in the report?*
Interviewee: *Do you mean, why it fails your check?*
John: *Yes exactly, do you know why?*
Interviewee: *Hmm no I'm not sure, let's see if I can open it and take a look. I just do not remember straight away... So basically you comment some check and we only assert true or assert false and it works, pass and not fail.*
John: *Yes so the thing I am doing is that I'm in this method I am taking the entire body of the inflateItem method and replacing it with a single return. Then I run the test suite. If every test still passes, then I report it here in the report.*
Interviewee: *So you comment inflate in this line? I'm just trying to under-*

stand exactly what you comment here.
John: *Yes, so lets see. Let me share the tab with the code as well*
Interaction: *Showing code from PR on GitHub*
John: *The thing I'm doing here is I'm replacing the method of by removing all these highlighted lines and replacing them with a single return so the method does nothing. I then run the test suite for the module. And when I do that, all test cases still pass, so that's why I've reported this as an error. Do you follow or was it unclear?*
Interviewee: *I'm trying so. So you comment and replace with something not in test classes? But in like, class which the test use.*
John: *Exactly, so the idea is that when you publish a pull request, then I see that you've changed some line within this method. Then the thing I do is that I replace the method to be empty to be empty, run the test suite to see if the test case passes. You can it as a way of evaluating the test suite to see if it tests the method properly and so on.*
Interviewee: *Okay I see. It is hard to say really. It is adding empty inside base classes. I mean I need to check it is hard to say right now. But because it is a void method it won't return anything. It is like inflate at view so it's like if it doesn't inflate anything it should be an error like a NullPointerException or something like that but if it not throws any errors it means that is was inflated. It could mean that the tests are wrong or that we do not cover that specific case. Or maybe somewhere in base classes they handle this cases in some way by not throwing or something. I don't know really.*
John: *Do you think there could be a way of fixing this? Maybe by asserting something extra since it is a void method?*
Interviewee: *I think yeah, it would be good to write some tests and at least check this and see how it will be in real operations.*
John: *Okay so I mean you could try this locally by simple replacing the method to see how it works. You do not have to do this but it is totally possible. Anyhow, is this error something you have or plan to fix or do you wait for this? Is it urgent so to say?*
Interviewee: *It is not urgent, I would say. We just don't put a lot of tests for this package. We just started working with this feature and there can can be a lot of functionality that can be changed or removed or updated in the near term. And we have deadlines so for now we will not cover all classes by tests. It is like basic functionality so I think there will be fixes to this anyway. But I cant say there will be a fix this sprint, maybe in a future sprint. But not right now.*
John: *Can you say like, this is a non critical part of the code or is this part*

of the code different than other parts? I'm basically after things I can use to develop the tool further in terms of what we should mutate etc. Or is it because you have tight deadlines right now?
Interviewee: *I would say this is a critical part. I mean it is a base part which inflates a view. That is it adds a view to the screen. If an error occurs you would not see that UI element on screen. So it is like base functionality of this feature. One of the base functionality.*
John: *So, moving on. In the context of this report. I don't know how much you used the concept of GitHub Checks. But basically you can have the status as neutral, which means that you can just move ahead without doing anything on what was said in the report. Or I could have it so that you would have to acknowledge that you have an error in the pull request to press a button like, yes I accept that there is an error here. Or the third case is that you can have it as breaking, you have to fix the presented error before you can merge into master? Do you have any opinion on how you would like to have it?*
Interviewee: *It is hard to say right now. I would think the second and third variant. I mean if there is an error like that maybe it is not good to merge at all. Or at least accept to merge and now that you could fix it later with tests. But I am not sure. In general we should not merge without any tests.*
John: *So just like, did you see this report before I reached out to you over Slack?*
Interviewee: *Yes I saw this one. But I didn't have the time to check it properly*
John: *When I showed this to you, were you able to reproduce it locally or did you not try to do that?*
Interviewee: *I can try to reproduce it locally of course*
John: *You don't need to it was just a question if you tried it while looking at the report?*
Interviewee: *Oh no, I didn't do that*
John: *Also, some context questions regarding the report. So like, do you think that GitHub Checks is a suitable format for having this types of report or would you want it somewhere else?*
Interviewee: *Oh no I like it in general. It's good to have this tools for how your tests are behaving so I think it is good.*
John: *On the same note, do you think it could be presented in another system in CI or in the pull request or do you think this is the place?*
Interviewee: *But I thought we already had some test checking on the CI?*
John: *Like this type of tests or another one?*
Interviewee: *I'm not sure it's this type of tests or not but in general, like if your build break someones tests or your test don't pass, then you are not able*

to merge to master. So we already have some checking. I am not really sure how they are all connected to each other though.
John: *Yes so there are a lot of tests being run in the CI system today. So it already is like a validation place where you validate all your changes basically. They report the errors a bit differently. Some of them are bots that are commenting on the pull request.*
Interaction: *Showing a PR with comments made by tooling.*
John: *Some of them are commenting like this right here. Showing the errors. There are some different approaches in how you can present errors like this.*
Interviewee: *Okay cool*
John: *Going back to the report. Looking at this now and when I sent it to you, was there anything you would like to add to the report to maybe make it easier to see what happen or what this is?*
Interviewee: *Yeah for example, for me it wasn't clear inflatedItem which method exactly was replaced. I first thought it was about test classes. Not just a regular class. So maybe somehow clarify in which classes exactly and which lines or something like this. Maybe also like a step by step on how to reproduce this issue.*
John: *Okay that's a good point. Like have a way of how to reproduce it locally.*
Interviewee: *Oh yes something like that. To understand.*
John: *That's an interesting point. Okay so what you could have is like since this is just a markdown report, you are limited to the markdown format. You could definitely have reproduction steps in the markdown report, but as in the earlier question, would it make sense to redirect you to a website or something where I could give you a richer report with some interactivity?*
John: *Would it be easier to understand maybe if I could show some more stuff?*
Interviewee: *I would say if you show more stuff it would be more understandable.*
John: *Anything else you think would be interesting over showing reproduction steps?*
Interviewee: *No, I think that is all.*
John: *Okay, would you like to remove anything from this report?*
Interviewee: *Nope. I don't think so.*
John: *So in general with this type of concept of pseudo-tested method. Is there any part of the codebase and so on that you would not like to perform these types of test on?*
Interviewee: *It is hard to say for me. I am not guru in testing. So I am not sure*

on how it will move data, request things and so on. I don't know. In general. I am not sure if I can give you a good opinion on this.
John: *So I've heard from some others, just to raise what I've heard. Do you have parts of the code that is not used in production like debug builds or some part of the code that is not critical. Do you have that sort of separations in the code? Would it make sense to have that kind of separation on where we run the tool or not?*
Interviewee: *No I don't have an opinion on this.*
John: *The final question is basically really open. Do you have any other suggestions when it comes to this concept? This report? Anything that could've come to your mind during this interview or while interacting with the report?*
Interviewee: *No I am sorry. Nothing to add here.*
John: *Okay cool then. That are my questions.*
Interviewee: *Happy to help!*

A.2 Interview 2

John: *Thank you. I think we can start by you just briefly describing your work and role within the project and what you do?*
Interviewee: *Alright. So my name is X and I work with a team called Multiplex which owns kind of the mobile client phasing parts of our product called Connect which basically allows you to connect to 3rd party speakers in the app. So it is like a middle-ground with some UI but also some background stuff as well. I'm the senior engineer in the team and I'm originally an iOS developer. But I've been doing more and more Android as well. But I'm also just interested in testing in general and have worked a lot with test certified program. And was there something else?*
John: *No I think that was a great intro! So I thought, the idea is to base this interview upon the report that I sent to you over Slack.*
Interaction: *Presenting screen and showing report*
John: *Basically there were two methods that we discovered to be pseudo-tested which was this, notifyAllDevicesLost that was replaced by return void and then the method notifyLostDevice that was also replaced by return void. That was discovered while testing. So the first question is basically like, what are the testing weaknesses of this presented method or presented methods in this case? Do you know why they fall through here?*
Interviewee: *Well I think like clearly some of these tests. Especially the ones that are expecting no action are not very robust tests. Because they are like,*

assert that I don't get any subscription updated. Then you like go ahead and like remove the code that never notify the observers. So like the test will pass. So we've actually been thinking about this like how do we ensure a test that kinda assert that nothing should happen. Like how do we more aggressively assert that it fails when the code. Basically how do we write the tests that would make it fail in this scenario. Yeah that's like kind of ongoing conversation within the team.

John: *Yes very interesting and I guess that would apply to the entire codebase?*

Interviewee: *Yes yes absolutely. Definitely. I think that there is a lot of tests. I think like the one that I looked at first was the actual one that, I sent it to my fellow Android developers as well. The test case... what was it called, shouldCallObserversWhenDevicesLost. Like, that thing passes I think that might be uncovering something really wonky.*

John: *That is interesting*

Interviewee: *Something even more than this case we talked about like asserting something not happening. But like this should clearly fail. Like, asserting that it does happen when you remove the body of the method. And that test passes, to me that is a clear indication that there might actually be something wrong with the actual code. Even more than the tests. So that was like the most kind of valuable insight.*

John: *That is interesting. So do you think like, this is just a question that comes to mind right now, so when you see this there is like a lot of test methods that actually exercises this method. Would it make sense to like present it in another way to showcase maybe the test cases that really are relevant to the method under test?*

Interviewee: *Yeah I think like. I know it is going to be hard to implement probably, but like, I think it could be more effective to try to highlight one test instead of the entire suite basically. It becomes a bit daunting.*

John: *Yes it is really daunting. And you know this is just a fun fact but PITest which is this mutation framework for Java that we use. It has these dependency distances so you can actually know how many jumps or method calls you are deep when you are testing a method. So I think it could be like visualized in some way.*

Interviewee: *Oh okay I think that would be really good actually.*

John: *Hmm cool, Moving on. Is this presented error something that you already fixed or plan to fix.*

Interviewee: *Yeah I think like plan to fix this. It is just a really busy time right now. I am definitely going to try this out the next time I pull an Android ticket.*

I've been writing internal RFCs and stuff. So I haven't, but I think we can mark this as like, will fix and like. Yes just like try to gate some sort of overall metric from to use this to like. We have this huge backlog of like fixing tech debt and we are actually working on a tribe initiative called ppx engineering practices. It kind of buys you a little time to work on tests that can be hard to work on normally. It can be hard to have your PM let you work on say refactoring a test class for something that haven't broke in a very long time. It would be very cool to be able to somehow like be able to get the overall health status of individual health classes or something. It might kind of be lost in the shuffle when it is just tied to pull requests. And in the test certified program that is actually something we did. Take a look at to try to boil down some sort of metric. But was more of regular mutation testing just like swapping booleans and stuff. But yeah I think, knowing about this, it is definitely something that we want to work on when we have a little bit of time to kind of sharpen our tests.

John: *That sounds really cool. I mean, so one of the issues I've had with this unfortunately is that it reports some false-positives. I've had to do some pre-filtering before I sent the report to you. The main issue here, from what I can tell is that PITest is basically instrumenting bytecode loaded into the JVM, because it is much faster than writing to disc. But Robolectric, the test framework is also instrumenting a lot of Android APIs and mocking them so there is some interference going on there and in some cases it produces false-positives. It should be solvable, but you have to spend some time to actually do it. Which is really unfortunate, it would be nice to have it correct on all cases.*

Interviewee: *Yeah and like as someone from the outside I imagine it is really hard to make sense of a lot of these tests that have like 200 lines each or something.*

John: *Yes it really can be, and especially when there are like 30 test cases as there are in this case. But it is really good to have this kind of feedback since that is what we are really after here. So, in this type of report, what would be the most appropriate CI-status for this problem do you think. So in this case it is neutral, I guess the other alternative is to have it as you need to acknowledge that you've seen this and you accept it going forward or the final alternative is to have it as breaking such that you have to fix it before merging into master. Do you have any ideas on this?*

Interviewee: *I think like, if we could get to a state were we can trust it to a 100 percent, and we can kind of keep it contained to your actual changes, I think like, being blocked on some tiny change because of tech debt. That can't happen really. So lets you are changing one line but this whole class has huge*

problems with bad tests. Maybe you have to fix a critical bug, etc. But if we could, I would be fine with being somewhat aggressive. If we can make sure that focus is on your actual diff. But I think like if it does nothing at all and we have like 60 things already I'm not sure people would actually notice it.
John: *Yes so that's is the thing. How do you notice this with the noise that can come with a pull request and do you want to notice it and so on. But it is interesting takes on it. And the idea to not break on critical fixes and so on is super good to..*
Interviewee: *Yeah but I think like, maybe you can just like try different things to see how developers react to it. I know that people usually can react quite negatively when they can't merge their code.*
John: *Yeah that is true.*
Interviewee: *I was just going to say that there is this fine line of like nobody knows that this exist or like just enraging everyone.*
John: *So actually just like a fun fact about this. A part of this work was inspired by some work that they've done over at Google where they use mutation testing in CI. They have a quite interesting approach there in their CI-status system where the reviewer is actually going through the errors.*
Interviewee: *Oh that is a pretty great idea actually.*
John: *Yeah and the reviewer can say like, please fix or this is not worth fixing. And they have entire architecture setup so if the reviewer votes not worth fixing on more than 10 percent of the errors, then the tool maintainers have to fix it to achieve higher relevance. So it's actually quite a neat setup for building these sort of tools in CI I think.*
Interviewee: *Yeah that is actually like a really good idea.*
John: *So like in some way how we can build a system like that with what we have. So moving on, do you think GitHub Checks is a suitable format for reporting this or do you want it somewhere else?*
Interviewee: *Just again like, how many checks do we have on Android now?*
John: *Yeah, a lot I think there are like ten*
Interviewee: *Yes that is maybe doable, but like, maybe it could be commented by a bot also like a info or something. But I'm not sure, maybe that is just ignored as well.*
John: *Yeah, cool. Okay, did you try to reproduce this locally or did you just look at it in the report page?*
Interviewee: *Just like created a quick test ticket. I haven't had the time to reproduce. I think I'm going to check this out more doing the hack days next week.*
John: *Cool*

Interviewee: *But I like looked at the code and like I said I've sent it to my Android developers and I don't even understand how some of these actually pass. So I think it's going to be pretty interesting to actually try this out.*
John: *Yeah it sounds like it. So, looking at the report again, is there anything you would like to add to it? We talked a little bit about like the test highlighting and dependency distances, but is there anything else you would like to see here?*
Interviewee: *I think like maybe a little bit of reasoning would be cool, just kind of in general. I didn't really understand why you picked this method. Was this method just picked at random?*
John: *Yes that is a good point. So I've picked. The thing I do is that I basically look at what lines the pull request is changing, from that I extract the methods that are changed based on that.*
Interviewee: *Okay cool, so just. Because it was like, hmm what was I changing. Oh yeah I was like messing with volumes and stuff so I was like devices lost, I was a bit confused.*
John: *That is a really good point, I could see that happening, like you change this one thing and then. It could be nice to actually visualize in some way how I infer that this was a method that you actually changed. Cool, anything else on that?*
Interviewee: *Just like again, feel like, because it seems that you've put a lot of work into this, but when I get over 40 methods, it feels a little like it is completly automated and like I tend to, my first thought was like, they can't all be wrong. Maybe they are but just like, trying to highlight something a little bit more. Even though it might not be fully correct, just like, could it lead to better results when dealing with developers..*
John: *Yes so I was actually like thinking how, in some way, how can I visualize this better. Because here you are obviously limited by the markdown format. One of things I thought about was if GitHub checks would be suitable. Maybe I could instead point you to a backstage plugin, a separate webpage. I don't know, but a place where I could visualize or provide interactivity in a better way. Where I can guide the developer somehow. But I mean that is obviously more complicated and you leave GitHub and everything.*
Interviewee: *Yeah it's probably going to, just some thoughts.*
John: *Yeah it's really good. So we've touched upon this as well but would you like to remove anything from this report?*
Interviewee: *Let me see, No I think it has what you need. I don't see what you would like to remove.*
John: *Cool. This is quite of an open question but I've encountered some of*

these where I don't mutate relevant parts of the code and presenting it to you. Are there any parts of the code that comes to mind that you would not want to run this kind of tool on?
Interviewee: *In our code, no like I think this should be pretty relevant for our entire suite. So no, nothing comes to mind there.*
John: *Okay interesting. Because I've heard some opinions like when you have a debug build, maybe it's not that relevant. In some cases I've been mutating auto-generated code with this autovalue classes.*
Interviewee: *Oh I still feel like, our unit testing should be asserting on what we expect even from auto-generated code. We do have a bunch of these auto-generated code from like logging and stuff like that but I feel that it would be good to put an empty string as a logging identifier to see if we are actually testing that we log the correct identifier. So I don't see for our auto-generated code any downside of not running this.*
John: *So like the final question then would be if you have any other final suggestions? For this tool in general, like any other thing you would like to add?*
Interviewee: *Just maybe like, trying to make it a little bit more quickly visualizable, using colors and I am assuming that this class in the report looks pretty bad with mutation testing. Like see some sort of quick red color or something. It's more like a rendering problem but I feel like being able to just see the code tied to the non-failing tests. But I think like the most important thing is to get people to actually try it out.*
John: *That is really good, so that is like one of the interesting things, like obviously, it's just a small part of the pull requests that have these kind of errors, so you want build up the habit of actually going into this page and checking if you've received a report. Like on the regular creation of a pull request, it just happens sometimes.*
Interviewee: *Yeah or maybe some sort of quick starts to try it out with some git diffs with the changed methods and which tests to rerun or something. And like, I'm like way more of an iOS developer. Maybe pure Android developers are quicker, they wrote bigger parts of the code. I kind of just need to get in there and start staring at the code since I didn't write much of it.*
John: *That's a good point and I got that in the last interview as well in if you could like show reproductions steps into how you can step by step reproduce this error locally. Like having a git patch that you could maybe apply locally and just have it visualized locally as well if you want so you can actually view it like that.*
Interviewee: *But overall I think this is a good initiative and I think have you talked with the guy who is an expert on Android development here at the com-*

pany .
John: *Yes, he helped me setup the tool.*
Interviewee: *Okay awesome, have a nice one*
John: *Bye.*

A.3 Interview 3

John: *Hi and welcome. So what I'm doing in this interview is that I will base it upon the report I sent to you last week. Discuss some things around that such as what can be improved and your eventual opinions on these kinds of tests and so on. But we could start with that you quickly describe your role and work within the project?*
Interviewee: *Okay, regarding this report then?*
John: *No more like a general overview of what you are responsible for in the Android project and your work there.*
Interviewee: *My name is x and I am an Android developer in a squad called y. We are working with experiments that we are hoping could drive some metrics. For example engagement and session length. This project I'm specifically is looking at now is how we can drive audio consumption up in the Spotify app.*
John: *Okay sounds really interesting. So I'm thinking that I'll start by sharing my screen so we can look at the report together.*
Interaction: *Showing report on screen.*
John: *Okay, so this was the report that I sent to you. First question on this would be, for this specific methods, what would be the test weaknesses on these methods?*
Interviewee: *Yes, as I remember we discussed, I think this is a false-positive. What I think is happening is that you are changing the auto-generated class and then, when the test is running, the class is recompiled and your mutation is overridden. That is my guess of what could be happening. I don't really now how I could run this without the AutoValue framework overriding the the modification. I haven't really dug much deeper into this but I'm pretty confident that this test is behaving as it should.*
John: *Okay I see. What I'm thinking here, is for example this equals method under test here. What the report is saying is that the method has been replace by return false and return true. But since it is only one test case that is exercising this method, then I'm thinking that at least return false or return true. Some of them should be missed, if there is only one case. But maybe I'm confused. Do you see what I mean here? Either we assert that they are the same*

or that they are not the same. I guess of course we could assert both cases in a single test case.
Interviewee: *Yeah so this is a test of another subproject. What we would like to see is that this event is updating or model with new data. And then deep down into the subproject code this boils down to a subproject test. But what I'm interested in is basically that my new model is containing the new object.*
John: *Okay I see and that is the interesting thing here and also this thing here that this highlighted test potentially is maybe on a little to low level. Jumping down like 100 steps into an Autovalue class somewhere.*
Interviewee: *But your thinking that you should have a test for not equals as well or?*
John: *Well I'm thinking that it could be that is what is complaining about. Is is now saying return true and return false. And I have no idea about if that is reasonable or not. That is just a thing that this tool is discovering. And that is a thing I saw here is that since how this tool work, mutating the code directly in the JVM, it's producing mutants on auto-generated code that you cant easily test locally. Which could become problematic of course. But then its just, I don't know if this is good or not, this is why I'm talking with you here, just trying to gather as much feedback as I can on the tool and how we can reason on these types of methods.*
Interviewee: *I mean in general, I believe this is a great initiative. I catch myself sometimes being bad and not following test driven development and starting to write the code and then the tests. Then all of a sudden all the test cases pass. Then I start to second guess myself and change the test to see that it is actually running. And there this is a great initiative to discover bad tests.*
John: *Yes that is kind of what this is here for. Like an evaluation of your test suite. Next question on this, is this something that you've fixed or plan to fix? Maybe not since its a bit unclear if this is a false-positive or not.*
Interviewee: *Yes I'm as said pretty sure that this is a false-positive. I've not been changing in the auto-generated code but in the test cases and it seems to catch the reported error. At least when I change it to what I want to test.*
John: *Okay so this is not highlighting the thing you want to test or what is relevant for you during testing?*
Interviewee: *No I think it should if we could be sure that the overrides are actually staying. Then I think the return false would fail the test here since there would be different models. Though for return true we do not have a test written for. And I don't know if it is relevant to go down on such a level that we test that we send in a model to see if the model contains another model. I mean it is just a question of how many tests you want to write really. I don't see that*

you would gain a lot from that. But I definitely think that if the override would stay, return false would create a failing test.
John: *Okay really good. So for this report, if we can assume that there exists no false-positives. What would you think would be an appropriate CI-status for this report. It is now neutral, you could also have it as that you have to acknowledge it, noting that you've seen the errors. Or you could have it as breaking, meaning that you have to fix it before merging into master.*
Interviewee: *Well I am strongly of the opinion that tests are running and revoking as soon as they are failing and as long as we can trust them. Depending of course on how long it takes to complete these kinds of test. I would like to say that these kinds of issues should be fixed before merging into mainline. But given that you can realistically run it as a part of the premerge-checks.*
John: *Okay so you are thinking that you should be able to run it locally?*
Interviewee: *Not really, that doesn't really matter. I think it more about if this would take so long time that every pr would have a runtime increase of 10 minutes, then that would be a consideration question. But I guess these sort of tests are not super expensive.*
John: *No this test performs kind of quickly since I only look at the methods that are changed, I look at the lines and then infer what methods have been changed. So there won't usually be a lot per pull request. So it comes by pretty quickly. It is absolutely not one of the slower jobs in the pre-merge task.*
Interviewee: *But then I would happy to see this running pre-merge and being a requirement for being able to merge into mainline. But of course it assumes that it does not produce false-positives.*
John: *No of course, that is interesting. How are you feeling like the equals method we discussed earlier. Would that still be interesting. Because now you were saying that in the case with return true if it was worth it since it does not exist any test case, but if you would have it as breaking, how would that look then?*
Interviewee: *No I dont really now really. Then you would expect the test case to fail when you replace it with true. I don't really now when return true would be reasonable. Probably only when you expect it to be a difference. And if it doesn't exist such a test case, then you cant draw any such conclusions in this case.*
John: *Then we can move forward. In this case we are using GitHub Checks. Do you think it is a reasonable format for this kind of report?*
Interviewee: *Yes I believe so. I haven't seen this kind of report earlier so I had to spend some time to actually understand it. But given that you've seen it once I do think that the information given is sufficient.*

John: *A follow up to that, could it be better to use any other format such as for example a website that could provide more interactivity, guiding the user through the report. Clearing up some of the unclear things. Especially with auto-generated code there could be some confusions.*
Interviewee: *No I like to have it as this. But this would come up as an issue on the PR? So it would probably be a lot easier to discover this than finding some errors nested deep withing a 14k Gradle output log trying to discover whats wrong. So I prefer this over that every day.*
John: *Okay cool, another question that is not super relevant here maybe is if you tried to reproduce this locally?*
Interviewee: *No what I did was to test to change the implementation that the test case implementation is running over to see if the test case failed, and it did. So I'm pretty sure that this issue could be incorrect. So in some sense I reproduced it, but of course I couldn't change the implementation of the auto-generated method.*
John: *Exactly and that is one of the things that are interesting here. Thinking about stuff like how do I reproduce this. It is an interesting exercise just thinking about how do I try this locally?*
Interviewee: *Yes I think given that it would not be an auto-generated class it would be pretty obvious what should be done.*
John: *Yes sure, to the actual report, would you like to add anything to it?*
Interviewee: *No I don't really think so. I would probably be able to provide some more input if we had some proper failing tests. But as a start I like this?*
John: *Okay is there anything you would like to remove?*
Interviewee: *No I don't think so.*
John: *Okay. Then I have a question which we've touched a little upon. Is there any parts of the code that you would not like to run this tooling on?*
Interviewee: *Well, as long as you can guarantee reliability I believe they will always be good to show that you don't have broken tests. Maybe the most relevant thing is old tests. I mean it's pretty probable that your test is correct when you are writing it, however, when you change the implementation but not the test case, then we could catch it. When the test passes but they don't test anything, it seems like this tool is made for that. I would think that is happening more often on old tests than newly written tests. But that doesn't mean that it is not relevant on new tests as well of course.*
John: *Okay interesting, well the only thing is like I've got some feedback such as some of these auto-generated methods are created by a library written by Google which we can assume are reliable. But you think it is good to test anyway?*

Interviewee: *Yes, just because autovalue is generating correct does not mean that the test you are writing is correct. That are two different things*
John: *Okay, then the final question is if you have any other suggestions or opinions about this?*
Interviewee: *No, not really. I think that this is really really good to have.*
John: *Yes it would be really nice to overcome the false-positive thing to be able to trust this to a 100%. This type of tool really depends on that.*
Interviewee: *Yes absolutely it is enough that it is one false-positive and people stop believing in it. I'm a victim of that myself.*
John: *yes me to. I guess it is just natural. Especially when it is neutral you eventually just stop to look. Thank you a lot for taking the time.*

A.4 Interview 4

John: *Hi and welcome. In this interview I plan to base it upon the report that I've sent to you over Slack. The one on your pull request. I plan that we look at that, discuss it and see if you have any opinions on it.*
Interaction: *Sharing report on screen*
John: *We could start with you explaining what you work on in the Android project and what your role is?*
Interviewee: *I work in a team that is responsible for the remote configuration sdk. We build an sdk that is living in another repo which we plug into all of the Android applications. We then make sure that the applications can collect the configurations and properties to be able to support a/b testing and other experimentations. It does some reports back to our experimentation platform as well. So in principle it means that we have some things that we own in my team, sdk is on such example, but we also help people across the android project in building integrations towards our sdk. In this case it was the automotive application that we helped refactoring to make it easier to integrate our new upcoming sdk. So I'm changing in their code and then you have to work tight with that team so they now what you are doing in the code and why. Just to make sure that a potential pull request will be approved later on.*
John: *Okay so you can potentially make changes all over the android project and help with integrations and such?*
Interviewee: *Yes exactly*
John: *Okay interesting, then I think it will be especially interesting to take a look at your view on this type of test since it won't neccesary be changes to code that you are the original author of.*

Interviewee: *Yes exactly, that is often how we do, there is often a lot of code and then we try to position ourself somewhere in the middle of some flow and then there is a big risk. For example this pull request was about their dagger graph was not supporting what we will be doing in our upcoming sdk. I thus needed to perform a quite large refactoring to prepare.*
John: *Okay that is interesting. If we look at the report, there is one method, getBrowseActionResolver, do you know what are testing weaknesses for this method that make it fall through?*
Interviewee: *I have as I said, which is potentially unlucky. I have no idea about what that method is doing. Well I can make a guess just from how they usually name stuff and so on in this team, but I don't know anything about this method per se. But in general, the app that use this method don't have any UI, instead there are data methods that populate built-in UIs. So I'm thinking that this method is responsible for populating some data regarding the browse view or something.*
John: *Okay cool, and I do think this is interesting since I'm trying to figure out how it would work to put this in the CI. When it is a project as large as this and there are people like you working more cross-functional over the entire project not necessarily having the detailed knowledge as the maintainers of the feature. It's interesting how you should frame your feedback with this setup. So it's really good to have this insight as well. So next question, is this something that you plan or someone else plan to fix?*
Interviewee: *No well what I'm thinking about the report is that we've found something where the test is redundant, it does not add any value. And I mean, that is scary in itself since it mean that we thought we had good coverage and had a false security on that. But in reality it turns out that is not the case. Then we have to look over our test cases and ask ourselves if they really provide the value we think they do. And that is a little bit how I like to think. You have to not write tests for the sake of it but rather more about what the tests deliver in terms of value and put that in relation to a risk/reward or something.*
John: *Okay nice. Next question about this. So now the report is presented as a GitHub Check, and the status of this report is neutral. The check is always passing and it is simply information. The other alternatives is to have it as acknowledgement. You have to accept the error but if you do it is fine to merge into master. The final alternative is to have it as breaking such as you have to fix the errors before being able to merge into master. Any opinion on the status on this?*
Interviewee: *I should probably recommend, with the premise of us having source code in a decent enough state. I'm thinking that when I write a new*

test when this tool is triggering, such as you wrote a new test that is not testing anything, then it should fail I believe.
John: *Okay so when you write new functionality with new tests then you make sure to fail the build?*
Interviewee: *Yes exactly. And then of course I also think that you should be able to override the tool such that I know better than the automatic tool. You should be able to make it pass anyhow. Maybe by putting an annotation in the code or something. But you should have to make an active choice of telling the tool that this should not be tested. But I absolutely think that we could fail on things like this. Now GitHub Checks is used as the report platform for this. Do you feel like that is a good place to have reports like this or do you have any opinions on another place to put it?*
Interviewee: *Honestly, I do not know how we ended up here. But I would like to have it as a comment thrown in my face on the pull request. Also even though the comment is not necessarily failing the build it could be interesting to get a comment like, btw do you know that your PR is doing this to blabla. I dont think that would hurt the process given that we don't have hundreds of these checks, spamming the PRs. Because know I guess you have to navigate to the Checks page to see what actual checks was being run on a given PR?*
John: *Exactly, and that's what can be a bit problematic with this format, especially when they are neutral, they can feel a little bit hidden. I mean there is a lot of different checks here that is quite informatic. So you have to click through them all and look through if you are interested. And I mean in some way it is related to the question about CI-status since that is actually forcing you into the page to look at the results. But of course, there could be better way to present it?*
Interviewee: *Yes well as they look know they are so many such that I don't look at them at least. All pass and I am safe. I should be able to merge know. It is the mentality I've been running.*
John: *Yes I understand that and I could probably be guilty of that as well. Because it is so much and you look for errors and it looks good.*
Interviewee: *Yes it gets hidden in the noise really.*
John: *Okay so did you see this report before I sent it to you over Slack or was that the first time you saw it?*
Interviewee: *I think I remember seeing something about it in the source code of the changes like annotations.*
John: *Yes it did, so with GitHub Checks you can annotate some of the code, however it is a bit limited and you can only use plain text there. Just allowing me to say there was an error there but not necessarily allowing me to explain*

everything. That still has to happen in the report. But yes it is possible but the functionality leaves some things to be wanted.
Interviewee: *Okay but I think I saw it as a comment and I think that is a pretty nice thing.*
John: *So that was a nice thing to have it as annotations in the code?*
Interviewee: *Yes I do think so*
John: *Okay cool. Did you try to reproduce this method locally?*
Interviewee: *Oh no I did not try that at all.*
John: *If we look at the report. Is there something you would like to add to it? That you feel is missing?*
Interviewee: *No I don't know. I get to know what was changed and updated. I get to know what tests and even links which is nice. And I get to see what replacement was made. So no I like it. Maybe something like click here to generate a JIRA ticket or something that is prefilled with some information regarding this. Just to make it easy to put this as tech debt for the team. To put it as I would like to fix this but I can't do it now.*
John: *Okay super interesting that you could create a bug report directly. To the contrary, anything you would like to remove?*
Interviewee: *Not really. The feedback buttons is probably not giving any value but I see why you have it there.*
John: *Okay. Then I have a quite open question. Is there any part of the codebase where you feel it is not worth running this tool on? Any rule you could apply for excluding certain PRs?*
Interviewee: *No I don't know. Not directly. Maybe some integrations with c++ libraries could be hard to do. But otherwise no.*
John: *Just a thought I got know. You talked about you working more cross-functional. Would that be any criteria, such as ownership and the person making the changes or is it still relevant to surface it to you and then you could create a ticket?*
Interviewee: *No I still think it is good. It should surface for all independent on who owns the code. Then it is more a little of a shame game such as then I know how the state of your code is.*
John: *Okay nice, anything else you want to add or any questions you have?*
Interviewee: *No. What I'm thinking about is now we have modules which are only Kotlin, have you excluded them specifically or?*
John: *Precisely so in the tool I'm using a framework called PITest which is for Java in how it statically analyzes the code in finding mutations. Right now it only works for Java so what I do in the CI tasks is filtering out all the Java files so unfortunately Kotlin files aren't being analyzed right now. I've ignored*

them in this project.
Interviewee: *Okay cool.*
John: *Thank you a lot. Bye.*

A.5 Interview 5

John: *In this interview I plan to base it on the report I sent to you last week, discuss it and see if you have any opinions on that and the tooling in general. But we could start with you briefly describing your work and role within the Android project.*
Interviewee: *I'm an Android developer, I've been at the company over three years and my team focus on podcast. We own the podcast show page and some other small functionalities regarding the podcast integration integration.*
John: *Nice. I will start by trying to present my screen and then we can look at the report together. This was an interesting example with the equals auto-generated method being a little bit confusing. But, the first question. What are the tested weaknesses of this presented method, if any?*
Interviewee: *You mean this one, specifically the equals?*
John: *Yes the equals.*
Interviewee: *Yes we talked about this briefly before but we very rarely test methods that are automatically generated because I think this is Autovalue. We don't test the methods that comes with the automatically generated class. But I guess we are, especially equals, interested in only asserting on the attributes. Rather than checking if the model is equal to another model. Do you understand what I'm trying to say?*
John: *Yes so you assume that the Autovalue framework is behaving correctly and then you basically assert on a higher level. Not on the equals level but on a feature level?*
Interviewee: *Yes exactly.*
John: *Okay cool. In this case, you would not need to add a negative assert or something. Second question is if this is something that you've fixed or plan to fix?*
Interviewee: *Not really in this case because as I said we don't see the value, atleast for the test case that was generated in the report because we do not use equals here anyway.*
John: *Okay so what happens here do you think, since you are generating a in some of these test cases, but you are never testing for equals so it is return true and return false that you are not checking?*

Interviewee: *Yes*
John: *So for this type of check, This is a GitHub check, and as you can see, the status is neutral meaning it always passes, line an information report. But there is the alternative to have it as acknowledgement needed, meaning you have to press a button to have it merged into master. Or you could have it as breaking meaning that you have to push a fix to the PR before merging into master. Do you have any opinions on how you would like this sort of report.*
Interviewee: *I think leaving us the decision if we want to fix it on the same pr is good. But I see value in this, not for generated classes but for classes that we generated ourselves. So if we have methods that for example is active and something that we created I see the value that when it returns false it should have an effect on other models and the state of whatever class you are testing. I think it is quite good to ensure. Because it will help us ensure that our test coverage is quite good. But I only see the value in that if it is for the classes that we wrote.*
John: *Okay so for developer written methods. And do you think the ack-needed status is nice?*
Interviewee: *I'm leaning towards that we need to accept it since it could potentially be false-positives. But when we can trust the tool even more that there are no false-positives I see value in having it as a pre-merge check that has to pass. That way we can ensure that the test coverage is better. But at this point we would probably not like to enforce it since there are a lot of checks already running.*
John: *Okay cool makes sense. So looking at the report again. This is GitHub checks, do you think this is a suitable place to report this kind of stuff.*
Interviewee: *I think I like it, because I've seen on some of the PRs that there is like a note in the changed code. Theres is like a note that this is a pseudo-tested method. I think that is good. But to me I click on it just to see. But probably because you've reached out to me and I've seen what it is about. But I am not sure how it is for other developers. But I think it is good to have those nodes interlaced with the changes instead of as a separate item.*
John: *That is actually really interesting thing with these GitHub checks. So I wanted to have those nodes better populated with what is going on. The thing is that you can only have plain text in them. So I was really longing for them to have markdown so I could show some more info. But it was quite limited in that way. So if that could be improved I think it would be really nice. Okay cool, moving on a little bit. Did you try to reproduce this locally? Or do you think it is even possible to do?*
Interviewee: *What do you mean like?*

John: *So for a developer written method, reproducing locally then you could simply edit the source code to just return true. In this case it is probably more confusing in how you should do that since this is generated at compile time.*
Interviewee: *Yes I don't know. If it is a pseudo-tested method that we wrote it would be pretty simple to do. It is hard because I guess the mindset that I have, for example we have a method that returns boolean. What I usually do is that I write a test case for when it returns false and a test case when it returns true. But I don't just replace the entire method to return false. Do you understand what I'm trying to say. We probably pass something into that method that we know should return false.*
John: *Okay. looking at the report again. Context wise, is there something that you would like to add to this report. To make it more clear or something?*
Interviewee: *I am not sure. Because I think now that we talked about this a bit it is a bit more clear to me. Because the part with the test cases it feels a bit unclear in what way does the test case affect this. And also the link was broken which makes it a bit hard to understand.*
John: *That makes sense, especially for these auto-generated methods it is quite confusing to actually know what is going on. If it would be developer written it would be more clear I guess. But now it is like these framework generated methods which we do something to.*
Interviewee: *Yes I think it would be good as well because I feel like there would be a lot of warnings about this that are really not errors per se. Developer sometimes have the tendency to just ignore everything.*
John: *Yes exactly and that is what I've discovered. And maybe as you've seen I've had some issues with false-positives on this tool and it really just highlights the importance of having a robust tool. Because developer lose interest so fast. And I'm a victim of that myself. Like if you can't trust the tool, you don't look at it. So that is the key concept here. Same question but reversed, would you like to remove anything from the report?*
Interviewee: *No I don't think so. It is good that there is a link to what pseudo-tested methods is. Probably the part about the feedback, the action items could be improved to like a button or something. For some reason I felt that I didn't have to do action on it. So I don't know what improvement can be done on that.*
John: *Yes so that is just basically me trying to gather data for my thesis. Trying to figure out how you think about it. We touched a little bit about this in the start of the interview. Are there any parts of the code that you don't want to run this tool on. More than auto-generated methods?*
Interviewee: *I can't think of anything. Probably I'm just constrained to the*

experience of this PR. But aside from autovalue stuff. Do you do this for private methods as well.
John: *Yes I do so it is interesting to hear your opinions on that?*
Interviewee: *It is hard to say, because I'm not sure what, I can't think on the top of my head what the direct effect would be if we are changing private methods. I feel like that is one thing that we rarely think about. Because what I'm trying to say is that when we write tests, it is mostly testing the publicly exposed methods and the state change that it does. But it is like, we are not specifically thinking about the private methods or whatever method that are not exposed by us. But then, if that is the thinking, maybe it is good to test the private methods as well, because one way or the other it will have some sort of effect somewhere. So I think it is probably good to run this tool on private methods as well.*
John: *Okay cool that is an interesting note. Because it is a bit contrarian since the unit testing philosophy is that you only test publicly exposed methods, and then here I come mutating private. But it is interesting. So rounding off, is there anything else you would like to add?*
Interviewee: *Not really. I think it is pretty good. I can see the potential in this but I think false-positives and auto-generated classes should be removed in my opinion.*
John: *Thank you bye*

TRITA -EECS-EX-2020:523

www.kth.se