



Andreas Schulhofer, BSc

# Android Mutation Testing using Pitest and Android Gradle Build Tools in the case of Paintroid

## Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Telematik

submitted to

**Graz University of Technology**

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Institute for Softwaretechnology

Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, July 2018

This document is set in Palatino, compiled with [pdfL<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub>](#) and [Biber](#).

The L<sup>A</sup>T<sub>E</sub>X template from Karl Voit is based on [KOMA script](#) and can be found online: <https://github.com/novoid/LaTeX-KOMA-template>

## Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

---

Date

---

Signature



# Abstract

Mutation testing is widely considered to be an effective analysis method for evaluating and designing tests. However, there is still need for a tool which enables mutation testing for local unit tests as well as instrumented tests and can easily be integrated into the Android build system. Problems arise due to the nature of Android and test execution of UI tests tends to be slow. This thesis deals with implementing a mutation analysis tool which is easy to integrate into the current Android build system as a Gradle plugin. It uses Pitest as an approved tool for mutant creation and analysis of local unit tests. Additionally, the developed system enables mutation testing on an emulator or a real device. The implemented tool is subsequently used to determine the test quality of the Paintroid project. Paintroid is a drawing and picture editing app for Android, developed by the Catrobat project at Graz University of Technology (Austria) and is a free and open source non-profit project. On the one hand, this thesis demonstrates that mutation testing can easily be integrated in the current build system. On the other hand, it reveals some problems concerning the execution of test cases which do not handle resources correctly. Additionally, ways to speed up the mutation process are presented through the usage of parallel test execution and intelligent skipping of mutants.



# Contents

<b>Abstract</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Testing Background</b>	<b>3</b>
2.1 Mutation Testing . . . . .	3
2.1.1 Fault-based testing . . . . .	4
2.1.2 Fundamental Assumptions of Mutation Testing . . . . .	5
2.1.3 Mutation Analysis Process . . . . .	6
2.1.4 Difficulties in Mutation Testing . . . . .	13
2.2 Mutation Analysis Tools for Java . . . . .	16
2.3 Testing Android Applications . . . . .	17
2.3.1 Instrumented Tests . . . . .	18
2.3.2 Local Unit Tests . . . . .	19
2.3.3 Instrumented Unit Tests . . . . .	20
2.3.4 Automatic User Interface Tests . . . . .	20
2.4 Test of Paintroid . . . . .	21
2.5 Mutation Analysis Tools for Android . . . . .	22
2.5.1 Proof-of-concept Mutation Analysis Tool by L. Deng, Mirzaei, et al. (2015) . . . . .	23
2.5.2 MDroid+ by Linares-Vásquez et al. (2017) . . . . .	26
<b>3 Android Build System</b>	<b>27</b>
3.1 Gradle . . . . .	27
3.2 Android Gradle Plugin . . . . .	30
3.2.1 Android Plugin Types . . . . .	31
3.2.2 Build Configuration . . . . .	32
3.2.2.1 Build Type . . . . .	32
3.2.2.2 Product Flavor . . . . .	33

## Contents

3.2.2.3	Build Variant . . . . .	33
3.2.2.4	Application ID . . . . .	36
3.3	Android Build Process . . . . .	36
3.4	Android Debug Bridge . . . . .	37
3.4.1	Query devices . . . . .	39
3.4.2	ADB Commands . . . . .	40
3.4.2.1	Shell . . . . .	40
3.4.2.2	Install . . . . .	41
3.4.2.3	Uninstall . . . . .	41
3.4.2.4	Clear . . . . .	41
3.4.2.5	Pull . . . . .	41
3.4.2.6	Run Instrumented Tests . . . . .	42
<b>4</b>	<b>Pimutdroid Gradle Plugin</b> . . . . .	<b>43</b>
4.1	Plugin Overview . . . . .	43
4.2	Mutation Analysis Tool . . . . .	44
4.2.1	Mutation Analysis Process . . . . .	44
4.2.2	Mutation Identifier - MUID . . . . .	47
4.2.3	Mutant Markerfile . . . . .	49
4.2.4	Mutators . . . . .	51
4.2.4.1	Conditionals Boundary Mutator . . . . .	52
4.2.4.2	Increments Mutator . . . . .	53
4.2.4.3	Invert Negatives Mutator . . . . .	53
4.2.4.4	Math Mutator . . . . .	53
4.2.4.5	Negate Conditionals Mutator . . . . .	54
4.2.4.6	Return Values Mutator . . . . .	54
4.2.4.7	Void Method Call Mutator . . . . .	55
4.2.4.8	Constructor Call Mutator . . . . .	56
4.2.4.9	Non Void Method Call Mutator . . . . .	56
4.2.4.10	Remove Conditionals Mutator . . . . .	57
4.2.4.11	Inline Constant Mutator . . . . .	57
4.3	Plugin Configuration . . . . .	57
4.3.1	Install Plugin . . . . .	58
4.3.2	Configure Plugin . . . . .	61
4.4	Plugin Report . . . . .	65



## Contents

<b>5</b>	<b>Implementation Details</b>	<b>73</b>
5.1	Structure . . . . .	73
5.2	Mutant Creation . . . . .	77
5.3	Mutant Test Execution . . . . .	80
<b>6</b>	<b>Conclusion</b>	<b>85</b>
	<b>Bibliography</b>	<b>91</b>



# List of Figures

3.1	Typical Android build process . . . . .	38
4.1	Pitest Coverage Report Paintroid . . . . .	69
4.2	Pitest Coverage Report Class Mutation View . . . . .	70
4.3	Pitest Coverage Report Class Mutation Summary . . . . .	71
5.1	Pitest Export Folder Structure . . . . .	78
5.2	Pimutdroid Mutant Output Folder Structure . . . . .	80



# 1 Introduction

Mutation testing is widely considered to be an effective analysis method for evaluating and designing tests for various programming languages and other fields of application. In recent years, research also applied this method to Android applications to assess the quality of a test set. Due to the nature of Android applications and that tests can be run on an emulator or a real device in addition to the Java Virtual Machine (JVM), traditional Java mutation analysis tools can not be used. Various tools have been proposed by the research community, each approaching the problem in a different way. Despite the different approaches to mutation analysis for Android, it is considered to be an effective method for detecting faults in Android applications.

However, there is a lack of tools for Android applications which allow an easy integration into the current build system and also support the whole process of mutation testing. To address this gap, a plugin for the build system was created. It utilizes the provided tools from the Android Gradle Build tools to enable mutation testing for Android projects.

The core of this master thesis is the implemented mutation analysis tool Pimutdroid which has been used to determine the test quality of the Paintroid project. Paintroid is developed by the Catrobat project at Graz University of Technology (Austria) and is a drawing and picture editing app for Android. This thesis does not apply Android specific mutation operators or evaluate how well traditional mutation operators simulate real faults in the Android domain.

The thesis is organized as follows. Chapter 2 provides an overview of mutation testing, its fundamental assumptions, the traditional mutation analysis process, the different types of Android tests and presents the current mutation analysis tools for Android. Chapter 3 describes the Android build

## 1 Introduction

system with focus on the Gradle build system and the Android Gradle plugin. Chapter 4 describes the implemented mutation analysis tool and a deeper insight of the plugin is provided in chapter 5. Finally, chapter 6 concludes the thesis and suggests future work.

## 2 Testing Background

Chapter 2 provides an overview of mutation testing in section 2.1, it covers the fundamental assumptions of mutation testing, the typical analysis process and difficulties which arise using this technique. The result of this thesis provides a mutation testing framework which can easily be integrated into the current Android build system. For the implementation details of the proposed framework see chapter 4. The developed tool makes use of an existing mutation framework for Java. Therefore, in section 2.2 mutation analysis tools for Java are discussed. Section 2.3 provides an overview of testing of Android applications and section 2.4 presents a summary of the tests used in the Paintroid project and briefly explains the project. In section 2.5 related work on Android mutation analysis tools are discussed.

### 2.1 Mutation Testing

Mutation testing is a fault-based testing technique to determine the quality of a test set. This technique is implemented by seeding faults into the program and verifying if the test set was able to detect the injected faults. Each injected fault is a simple change to the original program. These faults either try to simulate an error that could stem from a programmer or are chosen from a set of errors which are typically for a specific domain. Mutation testing provides a test criterion which can be used as a test requirement for a collection of tests (A. Jefferson Offutt and Untch, 2001a,b).

How well a test criterion is satisfied is often measured as a coverage value. The test criterion or coverage criteria is defined by Ammann and Jeff Offutt (2008) as a set of rules which can be used to enforce test requirements on a test set. Test requirements are defined by Ammann and Jeff Offutt (2008)

## 2 Testing Background

as specific “things” that a test case or test set must satisfy. This supports the definition of properties which a test set should have. For example, the branch coverage criteria states that each branch of a control structure within the program must be executed. For each branch statement such as an *if-else* or a *switch* statement, the true or false outcome must be covered by the test set (Myers, 1979). Hence the requirement for branch coverage is to execute each branch of the program. For mutation testing the requirement imposed on the test set is “finding the injected faults” by killing mutants (see section 2.1.3).

### 2.1.1 Fault-based testing

In fault-based testing the quality of tests is measured by how many pre-specified faults are found by the test set, showing how many faults are not in the program (Morell, 1990). The measured quality can be used as an indicator to determine the strength of the test set or test strategy which is used to find faults. Myers (1979) for example states that a successful test case is a test that detects errors in a program and a test case that does not find an error is a waste of time and money. Morell (1990) states that fault-based testing takes a different approach to the traditional test statement by Myers, as successful tests indicate the absence of pre-specified faults in fault-based testing. This absence of pre-specified faults is a stopping criterion for testing, and faults which were not found highlight an area in the code which needs further investigation and testing.

Each pre-specified fault is used to create an alternative of the original program. The test data should distinguish the alternatives from the original program (Morell, 1990). Morell assigns mutation testing to bounded fault-based arenas, since there exists a set of alternative programs which is finite. Morell (1990) states that fault-based testing requires two conditions to show that a program is correct. First, the fault-based arena must be *alternative-sufficient*, meaning the arena contains a correct program and secondly the alternatives are not *coupled*. Coupled implies that a combination of alternatives is not detected by a test, whereas the single alternatives are detected by the same test. The assumption that mutation testing is alternate-sufficient refers to the *Competent Programmer Hypothesis* (CPH) (Morell, 1990).



## 2.1 Mutation Testing

This hypothesis was first mentioned in a paper by DeMillo, Lipton, and Sayward (1978) in 1978 and states that the program under test is close to a correct program. In section 2.1.2 the fundamental assumptions of mutation testing are explained in more detail.

### 2.1.2 Fundamental Assumptions of Mutation Testing

As described, the theory of mutation testing is based on two basic assumptions (DeMillo, Lipton, and Sayward, 1978), the *Competent Programmer Hypothesis* and the *Coupling Effect*. They were first introduced in a paper by DeMillo, Lipton, and Sayward (1978) in 1978. This paper is often cited as the seminal reference on mutation testing.

In the competent programmer hypothesis, which addresses programmer behaviour (DeMillo, Lipton, and Sayward, 1978), it is assumed that the program under test is written by a competent programmer, thus it can be assumed that the program is “close” to a correct version. Competent programmers have available two advantages: First, they have a rough idea about the errors which most likely can occur and second, that the program can be examined in greater detail while iterating through multiple steps in the process of creating a program. Therefore the program written by a competent programmer may be incorrect, but the errors it contains are relatively simple errors. The errors made by the programmer can be assumed to be small errors which can be corrected by small syntactical changes to the code. The written program differs from the correct program only by simple syntactical faults, mutation testing uses these faults and applies them on the program to create the set of alternate programs.

The other basic premise is the so called coupling effect. The coupling effect says that if the tests are capable of uncovering the simple faulty versions of the original program, then the tests are also able to detect more complex errors in a program (DeMillo, Lipton, and Sayward, 1978). Morell (1990) stated that “simple” and “complex” are not defined precisely enough to be verifiable, but that the coupling effect holds probabilistically under many circumstances. Later the coupling effect was supported by A. Jefferson Offutt (1992a,b). A. Jefferson Offutt precisely defined simple and complex

## 2 Testing Background

faults. According to A. Jefferson Offutt (1992a) simple faults can be fixed by making a single change to the code, whereas this is not possible for a complex fault. A. Jefferson Offutt extended the coupling effect by defining the *Coupling Effect Hypothesis* in the following way: “Complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults.” (A. Jefferson Offutt, 1992b). This general statement was restricted to mutation testing defining the *Mutation Coupling Effect* as follows: “Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage of the complex mutants.” (A. Jefferson Offutt, 1992b). Simple mutants represent simple faults and are created by applying a single syntactical change to the source code, whereas a complex mutant contains multiple changes. Another name used for complex mutants is *high-order mutants*. For example a second order mutant is a mutant that is created by combining two first order or simple mutants. Later Tai (1995, 2000) demonstrated the coupling effect theoretically.

The validity of the coupling effect was later examined in many studies (Andrews, Briand, and Labiche, 2005; Andrews, Briand, Labiche, and Namin, 2006; Daran and Thévenod-Fosse, 1996a,b; René Just et al., 2014). The studies generally showed that mutants are a valid substitution for real faults, but also that there exist faults which can not be simulated by mutants and that a reduction of tests based on mutation testing could lead to a reduced real fault detection (René Just et al., 2014).

Only if both the competent programmers hypothesis and the coupling effect hold, mutation testing or fault-based testing can guarantee fault detection. The traditional mutation analysis process is described in the following section 2.1.3.

### 2.1.3 Mutation Analysis Process

In the traditional mutation analysis process simple mutants are used to analyse a test set. The alternate programs of the original program are created by applying mutation operators to the program. Such an operator is also known as a *mutator*, *mutant operator*, *mutation transformation* or *mutation*

## 2.1 Mutation Testing

*rule* (A. Jefferson Offutt and Untch, 2001a,b; Wu et al., 1988). They are transformation rules which are applied on the original program to create one or more faulty versions of it, which are the mutants of the program under test. The operators can be seen as a formal description of a program transformation to a faulty version (R. Just, Kapfhammer, and Schweiggert, 2012).

The survey by Jia and Harman (2011) on the development of mutation testing shows that there exist a wide variety of mutation operators for different programming languages like Fortran, C, C#, Java or SQL. And that the mutation analysis process is also applied to other fields of application such as Security Policies or Web Services.

For example a mutation operator for the programming language Java is the Relational Operator Replacement (ROR) operator, for example available in the mutation system *MuJava* by Ma, Jeff Offutt, and Y.-R. Kwon (2006). A relational operator in Java takes two operands, compares their values and determines the relationship between them. The operator evaluates for equality and whether the first operand is lesser than or greater than the second operand. The six relational operators provided by Java (2018a) are shown in table 2.1. The ROR operator replaces one relational operator with another one and additionally replaces the whole predicate with true and false, so that the outcome of the evaluation of the operator is always true or false in the program execution (Ma and Offut, 2006).

Relational operator
$op_1 < op_2$
$op_1 \leq op_2$
$op_1 > op_2$
$op_1 \geq op_2$
$op_1 == op_2$
$op_1 != op_2$

Table 2.1: Shows the six relational operator provided by Java (2018a).

Listing 2.1 shows a method of a program which is using Javas unequal relational operator to compute its return value. The method checks whether

## 2 Testing Background

the two passed values are unequal, and if so, the first value  $a$  is incremented by one returned to the caller. If both values are equal the method returns zero. In listing 2.2 the method is shown after the ROR operator was applied to the program. The unequal operator was replaced with the equal operator (highlighted in red). This syntactical change created a new version of the original program that now contains a pre-specified fault which was introduced into the program.

---

```
public int logicZero(int a, int b) {  
    if (a != b) {  
        return a + 1;  
    }  
  
    return 0;  
}
```

---

Listing 2.1: Shows a method using the unequal relational operator. The method checks if the passed values are unequal and returns the first value  $a$  incremented by one to the caller. If both values are equal the method returns zero.

---

```
public int logicZero(int a, int b) {  
    if (a == b) {  
        return a + 1;  
    }  
  
    return 0;  
}
```

---

Listing 2.2: Shows the method from listing 2.1 after the unequal operator was replaced with the equal operator (in red) by the Relational Operator Replacement (ROR) mutation operator.

As described in section 2.1.1 the test set is now used to detect the mutated

## 2.1 Mutation Testing

version of the program. If the tests are able to distinguish the original program from the mutant, then the tests show that the pre-specified fault is not in the original program. Listing 2.3 displays two unit tests which verify the method called *logicZero* from listing 2.1. These two test cases are sufficient to show that the method is correct and both branches are covered as well as each program statement is executed. However the mutant from listing 2.2 created by the ROR operator will not be detected by the test set. The used input parameters of the second test case named *logicZero\_valuesAreNotEqual\_returnFirstArgIncremented* are not well-chosen and are not adequate to detect the change to the relational operator. Since the value of the parameter which will be incremented by the method is minus one, the return value of the method will be identical to the return value as if the passed parameters were equal. Hence, a potential bug introduced by replacing the unequal operator by a programmer would not be detected by the test set in listing 2.3.

Applying mutation analysis on the program would now show that the test set is not able to detect the fault introduced into the program. The undetected mutant highlights a location in the code base that needs further investigation. This allows a programmer to check the tests again and write and design better tests. To detect the mutant, the first input argument should be changed to a value which will not return the same result as in the case of equality. The adopted test cases are shown in listing 2.4.

The previous example depicts the typical steps performed in a mutation analysis process as described by A. Jefferson Offutt and Untch (2001b). A set of mutation operators is selected to create the mutants of the original program. The operators seed faults described by a transformation rule into the program. To determine if a test set detects a mutant each mutated program is built and it is run against the test suite. Prior, the test set is run on the original not mutated program and the outcome is stored as the expected result. This result is used to verify whether a mutant was detected by the test set. The mutated program likewise leads to a result after the tests are run against it. The actual result of the mutant is subsequently compared with the expected result. If the results differ, the mutant was so to say *killed* by the test set. A mutant is said to be *stillborn* if the program did not compile and was not even able to run against the test set. Should both the expected and actual results be the same, than the mutant *lived* or *survived*. This means

## 2 Testing Background

---

```
@Test
public void logicZero_valuesAreEqual_returnZero() {

    int firstValue = -1;
    int secondValue = -1;
    int expectedValue = 0;

    int actualValue = unitUnderTest.logicZero(firstValue,
        secondValue);

    assertThat(actualValue, is(equalTo(expectedValue)));
}

@Test
public void
    logicZero_valuesAreNotEqual_returnFirstArgIncremented() {

    int firstValue = -1;
    int secondValue = 0;
    int expectedValue = firstValue + 1;

    int actualValue = unitUnderTest.logicZero(firstValue,
        secondValue);

    assertThat(actualValue, is(equalTo(expectedValue)));
}
```

---

Listing 2.3: Shows two unit tests which verify the method in listing in 2.1. The test cases are enough to show that the method is correct and both branches are covered as well as each program statement is executed. However the test cases will not detect the mutant from listing 2.2.

```
@Test
public void logicZero_valuesAreEqual_returnZero() {

    int firstValue = -1;
    int secondValue = -1;
    int expectedValue = 0;

    int actualValue = unitUnderTest.logicZero(firstValue,
        secondValue);

    assertThat(actualValue, is(equalTo(expectedValue)));
}

@Test
public void
    logicZero_valuesAreNotEqual_returnFirstArgIncremented() {

    int firstValue = 4;
    int secondValue = 0;
    int expectedValue = firstValue + 1;

    int actualValue = unitUnderTest.logicZero(firstValue,
        secondValue);

    assertThat(actualValue, is(equalTo(expectedValue)));
}
```

---

Listing 2.4: Shows two unit tests which verify the method in listing in 2.1. The test cases are enough to show that the method is correct and both branches are covered as well as each program statement is executed. Additionally the adopted test case `logicZero_valuesAreNotEqual_returnFirstArgIncremented` will allow to detect the mutant from listing 2.2.

## 2 Testing Background

that the tests were not able to detect the fault which was seeded into the program. There can be two underlying reasons: The test set was not able to detect the fault or the created mutant behaves as the original program and is *equivalent*. Equivalent mutants are hard to identify and to automatically find all equivalent mutants is an undecidable problem (A. J. Offutt and Pan, 1996).

The measurement which is used to state how the test set performed is the *mutation score* or *mutation adequacy score*, it indicates the quality of the test set. It is the number of all killed mutants in relation to the total number of mutants. If a test set killed all mutants it is said to be *adequate*. If equivalent mutants exist and they were not sorted out beforehand, a test set is unable to be mutation adequate. The mutation score is either given as 0.00 to 1.00 or as in percentage from 0% to 100%. Goal of a developer or tester is to reach a mutation score of 1.00 or 100% by finding test cases which are able to achieve this goal (A. Jefferson Offutt and Untch, 2001b).

A. Jefferson Offutt and Untch (2001b) state that mutation analysis is an effective way to measure the quality of a test set. As a side effect the software is tested and it must be tested well or mutants will not be killed, as can be seen for the test cases in listing 2.3. The effectiveness of the approach is described by Geist, A. J. Offutt, and Harris (1992) in a fundamental premise: “if the software contains a fault, it is likely that there is a mutant that can only be killed by a test case that also reveals the fault.”. And L. Deng, Mirzaei, et al. (2015) state a source of the strength of mutation analysis as follows: “One source of that strength is that it does more than just apply local requirements, such as reach a statement or tour a subpath in the control flow graph (reachability), but it also requires that the mutated statement result in an error in the program’s execution state (infection), and that erroneous state propagate to incorrect external behavior of the mutated program (propagation)”. An example that this statement is true can be seen as well in the test cases of listing 2.3. In the example it is necessary to reach the mutant but it is not sufficient enough to kill the mutant.

Although mutation analysis is widely considered to be an effective criterion for evaluating and designing tests (Ammann and Jeff Offutt, 2008), there are a problems that will be discussed in section 2.1.4.



---

```
public int getLowerValue(int a, int b) {  
  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
  
}
```

---

Listing 2.5: Shows a method that will return the lesser value or the first value passed to the method if both are equal. Applying a ROR mutation operator which replaces the lesser than relational operator with a lesser operator will produce an equivalent mutant which can not be killed by a test.

### 2.1.4 Difficulties in Mutation Testing

Mutation testing suffers from a number of problems that prevent it from being widely used. For larger programs an enormous number of mutants will be created and for each mutant the whole test set has to be executed independently and the outcome has to be compared to the expected result. This leads to a very high computational and execution cost. This is especially true for Android applications due to the technical nature of how tests are written and executed for Android (see section 2.3).

Another problem are equivalent mutants. These mutants can not be detected by the test set as they behave identically to the original program. They are hard to filter automatically, as it is an undecidable problem to find all equivalent mutants automatically (A. J. Offutt and Pan, 1996). Therefore it often involves human effort to detect and remove equivalent mutants. Listing 2.5 shows a program method that, when the ROR mutation operator from section 2.1.3 will be applied to it, would produce an equivalent mutant. When replacing the lesser than operator with a lesser operator the result of the method will still be unchanged, thus a test will not detect and kill this mutant.

Additionally trivial mutants are mutants which are easily detected by the

## 2 Testing Background

---

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_display_message);

    Intent intent = getIntent();
    String message = noDecorator.decorate(
        intent.getStringExtra(EXTRA_MESSAGE)
    );

    TextView textView = (TextView)
        findViewById(R.id.textview);
    textView.setText(message);
}
```

---

Listing 2.6: An activity overriding lifecycle methods has to call its super class method, highlighted in red, in order to work correctly.

test set and do not add value to the analysis. Trivial mutants as a result increase the already high cost for mutation testing without aiding the outcome (L. Deng, Mirzaei, et al., 2015). For example in the case of Android an activity has to follow a pre-defined lifecycle and must invoke certain lifecycle methods in order to work. If an activity overrides for example the *onCreate* method it must call its super class method (see listing 2.6), otherwise the activity fails to start and will crash. If a mutation operator now removes this call, the mutant would always be killed when the test set is executed against the mutant on an emulator or real device.

Often redundant mutants are to blame for the high cost involving mutation analysis and they tend to misrepresent the mutation score and make it imprecise. Reducing the number of redundant mutants often increases the effectiveness and efficiency of the mutation process (R. Just, Kapfhammer, and Schweiggert, 2012).

Challenges involving mutation testing are the following:

- Very high execution cost due to the enormous number of mutants.

## 2.1 Mutation Testing

- Equivalent mutants can not be detected automatically and have to be filtered manually.
- Trivial mutants which do not add value to the mutation analysis process as they are detected easily by a test set. Hence they increase the computation cost.
- Redundant mutants decrease effectiveness and efficiency of a mutation analysis system (R. Just, Kapfhammer, and Schweiggert, 2012).

To this day, there exist approaches to reduce the computational cost of mutation analysis. A. Jefferson Offutt and Untch (2001b) divide these strategies into three approaches calling them “do fewer”, “do smarter” and “do faster”. The three types are further classified into reduction of the generated mutants and reduction of execution cost in the paper by Jia and Harman (2011).

A. Jefferson Offutt and Untch (2001b) define “do fewer” as the process to seek possibilities on how to run fewer mutants without reducing the quality of the outcome. The “do smarter” approaches try to reduce execution time for mutants or parallelize the execution. “do faster” approaches seek to generate and run mutants as fast as possible. Jia and Harman (2011) combine “do faster” and “do smarter” into the reduction of execution cost class and “do fewer” refers to the reduction of generated mutants class.

There are techniques which focus on reducing redundant mutants by viewing mutation operators not as atomic and they only use a subset of the operators to create mutants (R. Just, Kapfhammer, and Schweiggert, 2012). Kaminski, Ammann, and Jeff Offutt (2011) showed that the ROR operator creates redundant mutants when used in a traditional way and suggested the use of a subset to avoid redundant mutants caused by this operator.

The result of this thesis provides a mutation testing framework which can easily be integrated into the current Android build system. It makes use of an existing mutation framework for Java and is used to determine the test set quality of the Android Paintroid application. Therefore in the next section 2.2 mutation analysis tools for Java are discussed. The Android build system is described in chapter 3. Section 2.3 gives an overview of testing of Android applications and section 2.4 gives an overview of the tests used in

## 2 Testing Background

the Paintroid project. Related work on Android mutation analysis tools are discussed in section 2.5.

### 2.2 Mutation Analysis Tools for Java

This section gives an overview of mutation analysis tools for Java applications. As Android is mainly written in Java programming language a mutation system which is build for mutation of Java projects is used in the thesis to measure the test quality of the Paintroid project. In 2013, Delahaye and Bousquet (2013) compared a variety of mutation analysis tools for Java. The tools were examined based on their fault model, mutation criteria, features and performance when applied to different Java projects. Three usage contexts for these mutation tools were identified, namely teaching, research and industry. Table 2.2 shows the eight compared tools and their identified potential contexts.

Mutation Analysis Tool	Usage Context
Bacterio	research
Jester	-
Judy	research
Jumble	-
Javalanche	-
Major	research
MuJava	research
Pitest	teaching, industry

Table 2.2: Shows the eight compared Java mutation analysis tools based on their fault model, mutation criteria, features and performance when applied on different Java projects and their identified potential usage contexts (Delahaye and Bousquet, 2013).

From the mutation tools in the paper of Delahaye and Bousquet (2013), *Bacterio*, *Javalanche*, *Jester* and *Judy* do not seem to be under development anymore

## 2.3 Testing Android Applications

as of May 24, 2018. The latest version of *Bacterio* dates to 2012<sup>1</sup> (Mateo and Usaola, 2012; Mateo, Usaola, and J. Offutt, 2010). The latest changes to the *Javalanche* mutation analysis tool of Schuler and Zeller (2009a,b) were made in 2012 according to the commit history on the GitHub project (Schuler, 2009). *Jester* of Moore (2001) is available for download at SourceForge<sup>2</sup>, the latest version 1.37 is from 2005 (Moore, 2013). The *Judy* mutation framework by Madeyski and Radyk (2010) does not seem to be available anymore. And the latest version of *Jumble* was released in 2015 (Utting and Trigg, 2007).

The *Major* mutation framework is still under development and the latest version was released on May 31, 2017 (R. Just, Schweiggert, and Kapfhammer, 2011; René Just, 2018, 2014a,b). The latest commit on the GitHub project for *MuJava* by Ma, Jeff Offutt, and Y.-R. Kwon (2006) dates to August, 2016 (Jeff Offutt, 2015). *Pitest* by Coles (2018c) is currently at version 1.4.0 and development is ongoing as of May 24, 2018 (Coles, 2018e).

From the mutation tools only *Major* and *Pitest* seem to be in ongoing development. *Pitest* is an open source project available at GitHub<sup>3</sup>. It can easily be integrated in form of a plugin into existing prominent build systems for Java like Maven<sup>4</sup> or Gradle<sup>5</sup>. It is important to note that it works well with Gradle as the build system for Android builds upon Gradle (see chapter 3). *Pitest* can be used for Java application projects the are build with Gradle by using the plugin provided by Zajączkowski (2018).

The next section 2.3 gives an overview of the usual ways Android offers to test Android applications.

## 2.3 Testing Android Applications

This section and the subsection 2.3.1 to 2.3.4 are based on the Android testing documentation by Google (2018e) and the Android API reference

---

<sup>1</sup><http://www.alarcosqualitycenter.com/index.php/noticias/306-febrero-2012-nueva-version-de-bacterio-mutation-test-system>

<sup>2</sup><https://sourceforge.net/>

<sup>3</sup><https://github.com/hcoles/pitest>

<sup>4</sup> <https://maven.apache.org/>

<sup>5</sup><https://gradle.org/>

## 2 Testing Background

from Google (2018b) and give an overview of the different tests which are written to test Android applications. As testing for Android differs from testing traditional Java projects it is important for mutation testing to have an understanding of how tests are executed. The Android development environment offers its own testing framework which provides tools and APIs to build different types of tests and run them. Dependencies of the Test Support Library can be added to the projects' development environment. The test framework allows tests to be written in JUnit3 and JUnit4 style, whereby JUnit3 tests are to be considered deprecated in newer versions of Android. Therefore writing tests in JUnit4 style is the recommended way.

Typical Android tests can be grouped into unit tests, integration tests and user interface (UI) tests. Unit tests can further be divided into local unit tests and instrumented unit tests. Android additionally labels these test categories as small tests, medium tests and large tests. The test API offers Java annotations to mark instrumented test cases as small, medium or large. When executing the tests a tester can specify the test size which should be run. This supports grouping and execution of specific tests.

Small tests are typically unit tests which test a component in isolation of its dependencies and run fast. Medium tests are tests which integrate multiple components and evaluate whether they work together correctly. However, they do not test the full stack of an application and increase the confidence in the application that it runs correctly on an emulator or a real device. Behavior of external dependencies is typically simulated. Large tests verify the full application stack by testing along a common workflow.

For automated UI tests Android offers the Espresso testing framework and UI Automator. External UI testing frameworks can also be used, such as Robotium (2010).

### 2.3.1 Instrumented Tests

Instrumented tests are tests which need to be executed on an emulator or a real device. For their execution they need access to instrumentation information or real Android components. Therefore, the Android Application Package (APK) of the application as well as the test APK are built and

## 2.3 Testing Android Applications

installed onto an emulator or a real device. Instrumented test cases can be run with the default JUnit runner provided by the test framework (Google, 2010). A custom runner may also be used for test execution. The default runner named *AndroidJUnitRunner* can run JUnit3 or JUnit4 test cases as well as test classes which make use of test frameworks like Espresso or UI Automator (see section 2.3.4). The runner supports a variety of configuration options such as running all tests, all tests of a test class, multiple test classes or single test case.

### 2.3.2 Local Unit Tests

Local unit tests are test cases which are compiled and run in the Java Virtual Machine (JVM) on the local machine. Test source files must be stored under `src/test/java` when using the default project layout. This greatly reduces the execution time as it neglects loading and executing the tests on an emulator or a real device. Unit tests should test a single isolated unit such as a method, class or single component. External dependencies or dependencies which are other application components of the unit under test should be mocked. Their behavior can be controlled by a mocking framework like Mockito<sup>6</sup> or EasyMock<sup>7</sup> or by writing self made mock classes. Unit tests are not suitable for complex UI interactions and should run fast, so that the developer gets immediate feedback on code changes.

To be able to mock Android system APIs, Android provides a modified version of *android.jar* API file for local unit tests. This stubbed version contains no code and throws exceptions when parts of it are invoked. Thus, mocking frameworks can be used to mock Android components which are used by the unit under test.

Another way to execute unit tests on the local JVM, which depend on Android components, is to use the unit test framework Robolectric (2010). Robolectric emulates parts of the Android framework classes which can be run on the local JVM. Unit tests which need Android components can be

---

<sup>6</sup><http://site.mockito.org/>

<sup>7</sup><http://easymock.org/>

## 2 Testing Background

executed without the need for an emulator or a real device, hence reducing the need to write and maintain mock code.

### 2.3.3 Instrumented Unit Tests

Instrumented unit tests are unit tests which need to access real Android components or instrumentation information. Therefore they must be loaded onto an emulator or a real device. They are significantly slower compared to local unit tests (Google, 2018f). However they reduce the cost to maintain and write mock code for Android components or other dependencies, although mocks can still be used. Test code resides under a different location than local unit tests, as the instrumented test source files are stored at `src/androidTest/java`.

The APIs used in instrumented unit tests can be utilized to write medium test cases which integrate multiple application components in the test.

### 2.3.4 Automatic User Interface Tests

Automatic UI tests are instrumented tests which are executed like instrumented unit tests on an emulator or a real device (see section 2.3.3). They allow to simulate user interaction for an application. Test code files are stored at `src/androidTest/java`. For a single application the Espresso testing framework can be used to write automated UI tests. Espresso provides an automatic synchronization of test actions and waits before performing UI actions until a synchronization condition is met. The framework also allows stubbing and validation of intents within the application. For testing across multiple applications and Android system applications, the Android Test Support Library provides the testing framework UI Automator. UI tests typically test across multiple layers of an application or even multiple layers of multiple applications. They typically verify a common workflow which is offered to a user by the application.

The following section 2.4 gives an overview of the Android tests which exist in the Paintroid project.



## 2.4 Test of Paintroid

*Paintroid* or *Pocket Paint* is a drawing and picture editing app for Android (Pocket Paint, 2018b). It is a free and open source non-profit project developed by the Catrobat (2018) project at Graz University of Technology (Austria). The source code of Paintroid is available on GitHub (Pocket Paint, 2018a).

Table 2.3 shows the current test set of Paintroid. The test set consists of 41 Espresso test classes having 281 test cases, 17 instrumented unit test classes having 88 test cases and three local unit test classes having 18 test cases (as of July, 06 2018). To run all Espresso tests it takes on average 14 minutes on an average machine and on the continuous integration (CI) server Jenkins<sup>8</sup> used by the Catrobat project. The instrumented unit tests take about one minute and the local unit tests execute in under one second. It can be seen that Paintroid makes heavy use of Espresso tests in comparison to unit tests or instrumented unit tests. Of the 17 instrumented unit tests 10 test classes make use of Espresso to be able launch an activity and have therefore a much higher execution time. Previously the test cases which are now written with the Espresso API were written using the Robotium (2010) testing framework. The cause for nearly no local unit tests is the heavy use of global static objects throughout the application. These global objects and states make it hard to mock code parts and to set up a correct state which can be used for a local unit test. Currently work is done to get rid of these global objects and reduce calls made to these objects. Removing this hard coupled dependencies will allow to write more local unit tests in the future. Currently the distribution of test cases among the different test types form an inverse test pyramid (Cohen, 2009; Fowler, 2012).

The average execution time of tests is important for mutation testing, as explained in section 2.1 the test set has to be executed for each mutant. So the time to run the instrumented tests of Paintroid against four mutants takes approximately one hour, if executed consecutively. Important for mutation is also that test cases run stable, so there are no flaky tests involved in the testing process. Flaky or unstable tests will kill mutants without detecting them. This will falsify the mutation score for the test set as many mutants

---

<sup>8</sup><https://jenkins.io/>

## 2 Testing Background

Test Type	Test Classes	Test Cases	Avg. Execution time
Espresso	41	281	14 minutes
Instrumented unit tests	17	88	40 seconds
Local unit tests	3	18	430 milliseconds

Table 2.3: Shows the set of test for the Paintroid application. For each test type the number of test classes, number of test cases and average execution time is stated.

will be killed randomly. The execution time of Espresso tests is many times higher than the instrumented unit tests as an activity needs to be launched. In case of Paintroid some of the Espresso tests tend to be unstable and are failing from time to time because a view could not be found or artifacts in the application state from a previous test cases where present.

The following section 2.5 gives an overview of mutation analysis tools which are available for the Android platform.

### 2.5 Mutation Analysis Tools for Android

The in section 2.2 presented Java mutation frameworks including the tools from the paper by Delahaye and Bousquet (2013) are for Java projects. The tests for typical Java projects are executed in the Java Virtual Machine (JVM) on the local machine. The difference to Android projects is that for Android projects additional types of tests exist which must be run on an emulator or physical mobile device. Therefore, for each mutated version of the application an Android Application Package (APK) file has to be built in order to test instrumented unit tests or UI tests. The different types of tests are described in section 2.3.

To the best of my knowledge at the time of starting the thesis (November, 2016) the only available mutation analysis system for Android projects, which is able to run tests on an emulator or a real device, was *MuDroid* by Yuan (2016a). Additionally known was a proof-of-concept tool by L. Deng, Mirzaei, et al. (2015) which creates the mutants from Java code. It likewise supports mutation analysis for tests which need to run on an emulator or

## 2.5 Mutation Analysis Tools for Android

real device. In comparison *MuDroid* uses a different approach by creating the mutants by applying the mutation operators at Smali byte code level, instead of mutating Java source or byte code. Hence it does not need the original source code for the mutation analysis (Yuan, 2016b). In addition to these Android mutation analysis tools a Gradle plugin project for Pitest by Zajązkowski (2018) exists, as mentioned in section 2.2. For this plugin a forked and adopted version was available which enabled mutation testing for local unit tests by Pitest on Gradle Android projects (Wrotniak, 2018).

In the following section 2.5.1 the proof-of-concept tool by L. Deng, Mirzaei, et al. (2015) is presented. In August 2017 Linares-Vásquez et al. (2017) presented in a paper the mutation analysis tool *MDroid+* for Android applications. In section 2.5.2 this mutation testing framework is described. A part of the tool is available but it does not provide a whole test framework which builds, executes and analyzes the created mutants.

Based on the test cases used in the Paintroid project (see section 2.4), a mutation analysis tool was needed which could easily be integrated into the build system and which was able to run mutation analysis on local unit tests as well as on test cases which need to run on an emulator or a real device. The tool should analyze the result for each mutant and create a report containing the mutation score for the test set. The result of this thesis provides a Gradle plugin for Android projects which enables mutation testing for these kinds of tests. Since Pitest is in ongoing development and an integration for Android projects for the mutation analysis tool already existed, the created plugin utilizes Pitest and its Android Gradle plugin to create the mutants for an Android project. In addition to Pitest the mutants can be run against instrumented test cases. The implementation of the plugin is described in chapter 4.

### 2.5.1 Proof-of-concept Mutation Analysis Tool by L. Deng, Mirzaei, et al. (2015)

L. Deng, Mirzaei, et al. (2015) propose eight new mutation operators in their paper, which are specific to Android applications and they describe the implementation of a proof-of-concept tool which utilizes these Android

## 2 Testing Background

specific operators as well as traditional Java mutation operators. Parts of the MuJava (see section 2.5) mutation engine are used in the Android mutation analysis tool which implements these mutators. It supports 15 method level selective operators from MuJava (Ma, Jeff Offutt, and Y. R. Kwon, 2005), four deletion operators and eight Android operators. The tool creates the mutants from Java source code of the Android application. Additionally three of the eight Android mutators are Extensible Markup Language (XML) mutation operators which modify the Android manifest XML file (AndroidManifest.xml) and XML layout files of the applications user interface.

The proposed mutation analysis process in the paper consists of the following nine steps:

1. At first, the user selects the mutation operators which should be used by the tool. The user chooses from the 27 available operators.
2. First the tool executes mutation operators which work on Java source code. It applies them on the code and compiles each mutated source file to Java byte code class files.
3. Next the XML mutation operators are applied to the Android manifest and layout files, each mutator creates a modified copy of the original file.
4. For each mutated file generated by the engine an APK file is created. The stillborn mutants are not used in the result. These “stillborn” mutants do not compile and are discarded by the system. For XML mutants the original AndroidManifest.xml or layout definition is swapped out with the mutated file when packaging the APK.
5. Next the configured tests which should be run are packaged into the test APK file. The system supports instrumented tests written with the Android testing framework as well as tests from external test frameworks, such as Robotium (2010).
6. After compiling every mutant APK file, the original application is packaged as well and installed into an emulator or onto a real device. Additionally the test APK file is installed and the tests are executed

## 2.5 Mutation Analysis Tools for Android

against the original application. The test result is then stored as the expected result. It is used to identify killed mutants.

7. After creating the expected result, each mutant is installed into an emulator or onto a real device and all test cases are executed. The result for the mutant is stored as the actual result.
8. Afterwards the expected result is compared with each actual result. If they differ the mutant is marked as killed.
9. The mutation analysis process is finished by computing the mutation score as a percentage value.

The mutation analysis tool by L. Deng, Mirzaei, et al. (2015) does not implement any mechanism to detect equivalent mutants. They have to be evaluated manually as the tool does not help by identifying these mutants. Trivial mutants, these are versions of the application that crash on startup, are counted towards killed mutants. The proposed process differs from a traditional Java mutation process since it applies mutation operators to none Java source code or byte code files as well. By mutating the manifest or a layout file and it must package the application as an APK file and install it on an emulator or real device. The same applies to the test set which should be run. This has to be considered as well as the test execution time of UI test cases using Espresso or Robotium, as they are very time-consuming. As listed in section 2.4 for the Paintroid project the existing instrumented unit tests and Espresso test cases take about 15 minutes to run. The test set is executed for each mutant created by the mutation tool. The paper does not explain how the mutation analysis tool is implemented, nor how the APK files are packaged and loaded onto an emulator or a real device.

The set of eight proposed mutation operators for Android was extended by three additional operators in 2017 by Lin Deng et al. (2017a,b). In 2017 L. Deng, J. Offutt, and Samudio (2017) used their mutation analysis tool to evaluate if mutation testing is effective for Android applications. Their results indicate that it is effective. Six new mutation operators for Android were added to the set of the previous 11 operators.

## 2 Testing Background

### 2.5.2 MDroid+ by Linares-Vásquez et al. (2017)

Linares-Vásquez et al. (2017) presented a mutation analysis tool *MDroid+* for Android applications. The code which is used to generate the mutants is available online<sup>9</sup>. From a fault study 38 Android mutation operators were derived and are available in the tool. Only two of the eight proposed android mutation operators which were suggested in 2015 by L. Deng, Mirzaei, et al. (2015) overlap with the 38 operators defined by Linares-Vásquez et al. The complete MDroid+ analysis tool consists of MDroid+ that identifies and creates the mutants and a parallel execution architecture for running the mutants against the tests on Android emulators called the “Execution Engine”. For each mutant MDroid+ creates a mutated application which is compilable and can be packaged as an APK file which then can be installed onto an emulator or a real device. For each mutant it clones the whole target project and applies the single syntactical change on the source code or XML file (Moran et al., 2018). The available tool does not provide a way to transform the Android project to APK files nor a way to execute the test set against the mutants and evaluate the results in form of a mutation score.

---

<sup>9</sup><https://gitlab.com/SEMERU-Code-Public/Android/Mutation/MDroidPlus>

## 3 Android Build System

Chapter 3 gives an overview of the Android build system and the system it is based upon. Hence in section 3.1 the build management system Gradle (2018a) is described. Section 3.2 describes the Android plugin for Gradle, used in the build process of Android, which is described in section 3.3. Section 3.4 describes the Android Debug Bridge tool which supports connection to devices via command line and explains important commands which are used by the work in this thesis. The following chapter 4 describes the implementation details of the proposed mutation analysis tool using the technologies explained in chapter 3.

### 3.1 Gradle

Section 3.1 is based on the Gradle User Manual (Gradle, 2018c) and gives an overview of the important concepts of Gradle with regard to Android and plugin development. Gradle is a general purpose build management system that supports build automation for multiple programming languages and platforms. Gradle is structured into projects and tasks. A project is something that should be built like a JAR file or should be done like the deployment of a web-application. What and how it should be built is defined in the build script of the project. Gradle build scripts are script files written in code and can be created in either Groovy (2018) or Kotlin (2018) language. The build script is compiled on execution and uses the Gradle project API to drive the build process. Gradle builds consist of one or more projects, known as a single project build or a multi-project build. A project consists of tasks where each task is a single piece of work within the project. A project is basically a collection of tasks and a build executes these tasks in a certain order to build the project. A task can be an ad-hoc task that has

### 3 Android Build System

no predefined logic or an instance of a task class which can be configured to perform a predefined execution logic for different parameters. Multiple instances of the same task class can be added to the build. Tasks have a fully qualified unique name across all tasks within the build. This unique name is called the path. Likewise a project also has a unique qualified name. A task has a local name as well which is unique within the project the task is defined in. It can be used to identify and execute the task within the project. Tasks may have dependencies on other tasks in the build. Additionally tasks may be configured to run in a specific order, so that a task should run after another task or should finalize a specific task. Gradle ensures that each task will be executed only once and that tasks are executed in correct order. For that reason Gradle creates a directed acyclic graph (DAG) called the Task Execution Graph (TEG). The TEG is constructed and populated before any task is executed. This gives the possibility to hook into the build life cycle and configure tasks before their execution. For instance when the configuration of a task depends on the presence of other tasks that are scheduled to be executed or the configuration depends on certain build properties which are not known in advance. A task which is configured at execution can be configured in different phases of the build. For example a task can be configured before any of the scheduled tasks is run or after a task has finished execution. Another important configuration moment is when the TEG is ready. Hence all tasks that will be executed in the current build are known and configured. The graph manages the execution of the task instances and maintains an execution plan of the tasks which are part of the current build.

The Gradle build life cycle has three specific phases:

1. **Initialization Phase:** In this phase it is determined which projects are going to be built, those can be multiple in the case of a multi-project build. For each project an instance of the Gradle project API is created. Each instance will be configured in the next phase by executing its build script file against the instance.
2. **Configuration Phase:** The project instances created in the initialization phase will get configured and the build scripts are executed against the project instances. All tasks of the involved projects get created and



configured in this phase.

3. **Execution Phase:** To execute tasks, a single task or multiple tasks can be passed as command line arguments to Gradle. Gradle determines the subset of tasks which have to be executed from the whole set of tasks which are created and get configured in the configuration phase. Each task of the subset is then executed with regard to the dependencies and rules to order the tasks, as stored in the TEG.

To start a Gradle build from the command line it is recommended to use the Gradle Wrapper (2018). The wrapper consists of multiple files. It downloads a declared version of Gradle and executes the project build script for this specific version. This allows an execution independent of any installed Gradle versions on the machine. On installation of the Gradle Wrapper a shell script will be created as well as a batch file for Windows. These scripts should be used to build the Gradle project using the Gradle Wrapper, to ensure that the correct Gradle version is used.

It is possible to listen and respond to the build life cycle by implementing listener interfaces or using closures as event callback methods. An important notification is the event that the project has been evaluated. It allows to create tasks or configure tasks depending on project properties like extensions which are added by plugins or project properties which are passed as arguments.

Gradle supports plugins to add predefined tasks to the build of a project. The build system ships with a number of default plugins that can be added to a project. Like the Java Plugin which adds tasks to a Java project which allow for example to:

- compile source files,
- assemble a JAR file,
- test the code or
- generate the Javadoc.

Gradle plugins extend the capabilities of a project by typically applying a default configuration based on a predefined convention. Additionally

### 3 Android Build System

they allow the reuse of build logic across multiple projects which need the same or similar tasks to be performed. A plugin can be added to the project by applying it to the build script of the project. They can be added using additional build scripts that are linked into the projects build script. These *script plugins* can be applied from the local file system or remotely by specifying an HTTP URL. Another way to add a plugin is to use *binary plugins*, these plugins are classes which implement the Gradle *Plugin* interface. Binary plugins that are not shipped with build system are known as no-core binary plugins. They have to be resolved first. Binary plugins can be included from an external JAR file defined as a dependency to the build script or using the plugin portal by defining it in the *plugins* configuration block of the build script. Furthermore a binary plugin can be include using an inline class in the build script. Additionally the plugin sources can be placed under a specific folder named *buildSrc*, Gradle will look for this directory and compile the sources in it and will add the compiled plugin to the build script.

The work in this thesis is implemented as a binary plugin that can be added to an Android project (see chapter 4). A plugin has a unique plugin ID that is used to apply it to the build script. The next section 3.2 gives an overview of the Gradle plugin for Android and important configuration elements and concepts which have an effect on the created artifacts, task names and output directories. This is important to know for the provided plugin as it uses tasks of the Android Gradle plugin and the created output files.

## 3.2 Android Gradle Plugin

This section is based on the user guide by Google (2018h) on the Gradle plugin for Android. Android uses Gradle as the foundation for its application build system. The Android Gradle plugin adds Android specific capabilities such as a flexible build configuration, a variety of tasks for development, verification, release and default configurations. On the basis of Gradle it is easy to add project dependencies to an Android application such as external binaries or other Android library modules. They can be included from the local file system, a remote repository or as a separate local project module.

## 3.2 Android Gradle Plugin

Android offers Android Studio as its official Integrated Development Environment (IDE). Gradle is an integrated tool of Android Studio, however Android Gradle plugin and Gradle can be used independently from the IDE. Besides the Android Studio, an application can be built using the command line by utilizing the Gradle wrapper on a local machine or on a remote machine like a continuous integration (CI) server such as Jenkins <sup>1</sup> (see section 3.1). The output is always the same when using Gradle to build the project, whether using Android Studio or not.

A typical Android project is a Gradle multi-project. The build files are plain text Gradle build script files that use Groovy or Kotlin language to describe and to manage build logic. The Android Gradle plugin adds domain-specific language (DSL) elements to the build script which are used to configure the build. To use the plugin a classpath dependency to `com.android.tools.build:gradle:<version>` has to be set in the `buildscript` block of the root project. Where `<version>` is the version of the Android Gradle plugin that should be used.

### 3.2.1 Android Plugin Types

Different plugin types can be added to the build file of a sub project (Google, 2018d). The following plugins can be added by using the plugin ID belonging to a certain type of module:

- `com.android.application`: Configures the sub project to be an Android app module that builds Android Application Package (APK) files.
- `com.android.library`: Configures the sub project to be an Android library module that builds Android Archive (AAR) files.
- `com.android.test`: Configures an Android test-only module. Here the target project which should be tested is set using the DSL property `targetProjectPath`.

---

<sup>1</sup><https://jenkins.io/>

## 3 Android Build System

- `com.android.feature`: Configures the sub project to be a feature module. Builds an AAR or APK file depending in which module it is referenced.

### 3.2.2 Build Configuration

Android Gradle plugin gives the possibility to define custom flexible build configurations by using build types and product flavors. These are explained in the following sections [3.2.2.1](#) and [3.2.2.2](#). This section and its subsections are based on the user guide on build variants by Google (2018g). They allow to create different versions of the application by using a common code base and by adding additional code, resources and configurations for a specific version only.

#### 3.2.2.1 Build Type

A build type allows to define certain properties that are typically for a specific type of build. Gradle uses these properties when assembling and packaging the APK file of the application. They normally get configured for a different stage in a development process. By default the Android Gradle plugin creates the *debug* and *release* build types. Build types are defined in the `buildTypes` block of the Android Gradle plugin DSL. Neither *debug* nor the *release* build type must be present in the `buildTypes` block to be available, however they can be added to the block to change the default settings of the type.

Furthermore a build type has an impact on the names and the number of available tasks and on output directory names created under the build directory as well as names of created files. By default the APK file for running the instrumented tests is created for the *debug* build type, when running the corresponding task. By specifying the property `testBuildType` to the name of a build type, the name of this task changes and the test APK is built for another build type under a different output location.

### 3.2.2.2 Product Flavor

To create a different version of an application a so called *product flavor* can be configured. This allows to use additional or specific source code and resources for this version of the application, while still be able to share a common base of code and resources among the different product flavors. Product flavors are optional and must be created manually by adding the configuration to the `productFlavors` block of the plugin configuration. The `defaultConfig` block provides the default configuration for product flavors. Since Android Gradle plugin version 3.0.0 or higher a product flavor must be assigned to a flavor dimension. A flavor dimension assigns a product flavor to a logical group of flavors. This means that at least one flavor dimension must be defined. By default if only one flavor dimension is specified, each defined product flavor is assigned to this flavor dimension. Otherwise all flavors must be assigned to one flavor dimension manually by settings the property `dimension` in the product flavors configuration block.

Multiple flavor dimensions enable the combination of product flavors. Gradle combines for each dimension the product flavors of the other dimensions. Product flavors of the same dimension are not combined. Furthermore a product flavor and the flavor dimensions have, as the build type, an impact on the number or the naming of tasks, directories as well as names of created files by the build process.

### 3.2.2.3 Build Variant

Build variants allow to build different versions of an application using a single project. They result from the specified build types (see section 3.2.2.1) and product flavors (see section 3.2.2.2). Build variants are not configured manually. A build variant is the configuration Gradle uses to build the application.

The Android Gradle plugin identifies the available build variants by applying the cross product to the product flavors and build types. Any specified flavor dimensions also are taken into account. The number of build variants is equal to the product of product flavors per flavor dimension times the number of build types. For example using the configuration in listing 3.1

### 3 Android Build System

the product flavors *normal*, *premium*, *free*, *early* and *full* for flavor dimensions *base* and *status* will result in combination with the configured build types *debug*, *staging* and *release* in 12 build variants.

A build variant affects the naming of specific source sets, build tasks, output directories and output files such as the built APK file. The name of a build variant is built from the name of the product flavor finished by the build type. The build type is always applied after a product flavor. If multiple flavor dimensions are specified then each combination of flavor names resulting from the different dimensions is used instead of the single product flavor name.

The names of the build variants for the configuration in listing 3.1 are built of the following parts:

```
[free, early, full][Normal, Premium][Debug, Staging, Release]
```

For example this adds, to the application variants the Android Gradle plugin can build, the build variant `freeNormalDebug`. The task to assemble the applications APK file is named `assembleFreeNormalDebug`. The path of the output directory for the build is dependent on the build variant parts. The APK file will be stored under the output location `<apk-output-directory>/freeNormal/debug`.

---

...

```
flavorDimensions "base", "status"

productFlavors {
    normal {
        dimension "status"
    }

    premium {
        dimension "status"
    }

    free {
        dimension "base"
    }
}
```

## 3.2 Android Gradle Plugin

```
}

early {
    dimension "base"
}

full {
    dimension "base"
}
}

buildTypes {
    debug {
        testCoverageEnabled true
    }

    staging {
    }

    release {
        minifyEnabled false
        proguardFiles
            getDefaultProguardFile("proguard-android.txt"),
            "proguard-rules.pro"
    }
}

...

```

Listing 3.1: Shows a configuration using two flavor dimensions, five product flavors and three build types. The Android Gradle plugin combines these to create build variants which are used to build the application. 12 build variants will result from the two flavor dimensions, five product flavors and three build types. A build variant affects the name of the APK file, output directories and task names.

Undesired build variants can be filtered out by configuring a `variantFilter` logic block. Custom logic can be defined to ignore those build variants that make no sense or are not needed. The next section [3.2.2.4](#) describes the application ID and how it is affected by a product flavor.

## 3 Android Build System

### 3.2.2.4 Application ID

This section is based on the user guide for the Android application ID by Google (2018i). Every Android application needs a unique identifier to uniquely identify the application on a device. The naming rule for an application ID is in analogy to the Java package naming convention, but more restrictive. The application ID can be defined with `applicationId` property in the `defaultConfig` block. Every product flavor inherits the properties from `defaultConfig`, to change the application ID for a flavor, a suffix can be added to the base application ID using the property `applicationIdSuffix`. This property is also available for build types. Furthermore product flavors can overwrite the `defaultConfig` application ID using `applicationId` property in the product flavors configuration block. Here the ordering of flavor dimensions has an effect on the final ID. When the application ID is configured using `applicationId` in product flavors of different flavor dimensions, the base application ID is set to the higher ordered dimensions flavor. Suffixes are appended normally.

The ID of the instrumented test APK is by default set to application ID for the build variant and appended with `".test"`. It can be changed using the property `testApplicationId` in the product flavors configuration block or in the `defaultConfig` block. The test application ID is used to run the instrumentation test from the command line using the command line tool explained in section 3.4.

## 3.3 Android Build Process

This section is based on the user guide by Google (2018h) on the build process for Android. The general build process is shown in figure 3.1. Various tools and tasks are involved in the process of building the final APK file. In a typical application build first the compilers converts the source code and resources to Dalvik Executable (DEX) files and compiled resources as can be seen in figure 3.1. The java source files are compiled to `".class"` files using *javac*. The class files are then transformed to `".dex"` files using



## 3.4 Android Debug Bridge

the DEX compiler. DEX files contain the Dex byte code that is executed on an Android device.

For each build variant described in section 3.2.2.3, a corresponding task exists which compiles the Java sources to Java byte code. The build variants name is used in the task name, the generic name is `compile<buildVariant>Sources`. After the compile sources task the output files are located under `<build-dir>/intermediates/classes/<buildVariant>`. The task to transform “.class” files to DEX byte code uses the files from this directory as its input. This task has the generic name `transformClassesWithDexBuilderFor<buildVariant>`. The DEX files and compiled resources are then combined into a single APK file by the APK Packager. Using a debug or release keystore the packager signs the APK file. The application file must be signed first in order to be able to install it on an Android device. A debug keystore is automatically configured by Android, the sign the release version a keystore has to be created manually. After packaging and signing the application is optimized to use less memory using a tool called *zipalign*<sup>2</sup>. Following the APK Packager creates the final signed APK file which can be installed on a device. The file is either a debug or release APK.

Section 3.4 describes a tool which supports to connect and to run commands on Android devices. Additionally important commands that are used by the provided Android plugin for mutation testing are described.

## 3.4 Android Debug Bridge

This section and its subsections are based on the user guide for the Android Debug Bridge (ADB) by Google (2018c). ADB tool is command line tool provided by the Android Platform SDK. ADB allows to communicate with a connected device via a server process. ADB gives the possibility to execute device actions without the need to open a shell on the device. The tool has three components, an ADB client and a server which both run on a

---

<sup>2</sup><https://developer.android.com/studio/command-line/zipalign>

### 3 Android Build System

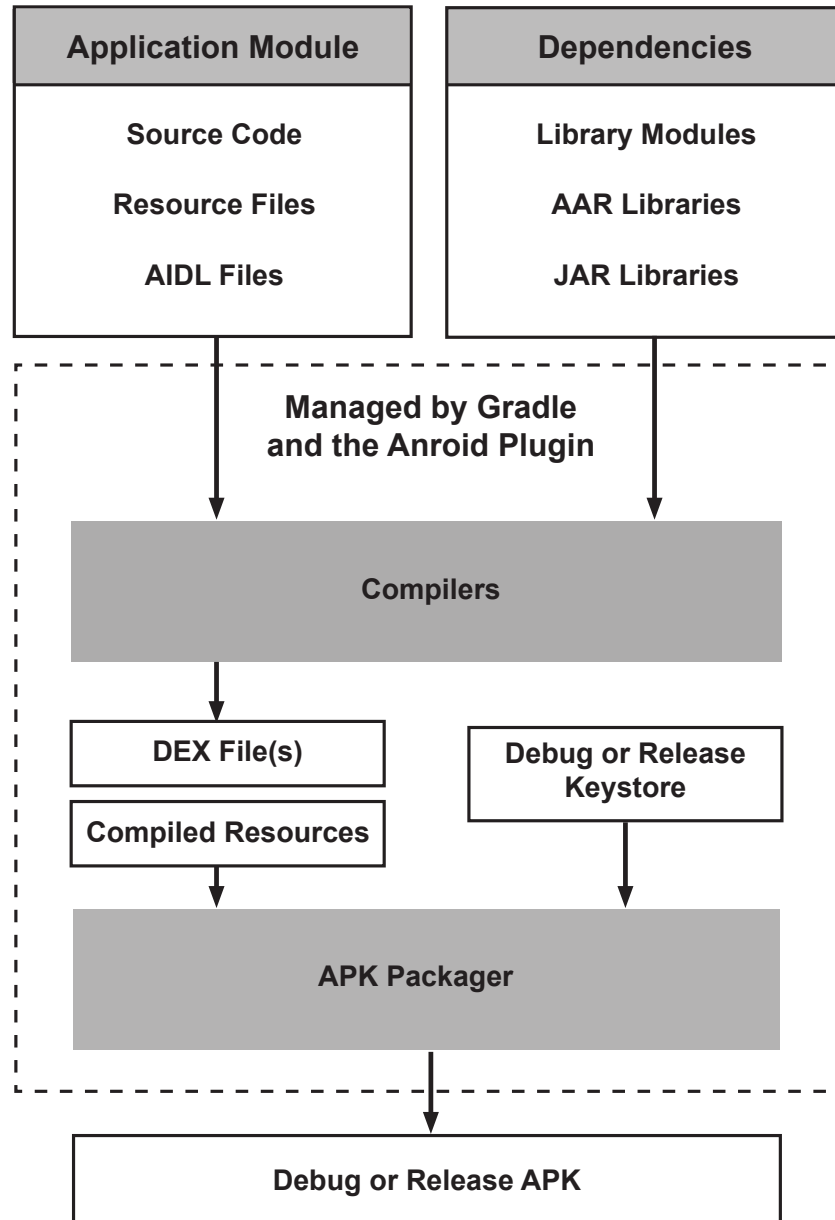


Figure 3.1: Google (2018). The typical Android build process using various tools and processes to build the final signed APK file from the application module and its dependencies. Note. Reprinted from: <https://developer.android.com/studio/build/#build-process>, Available at: [https://developer.android.com/images/tools/studio/build-process\\_2x.png](https://developer.android.com/images/tools/studio/build-process_2x.png), Accessed 20 May 2018.

development machine as well as an ADB daemon (ADB) that runs as a background process on a device. Through the client commands can be sent to a device, the server process manages the connection to a device and the daemon on the device executes the commands on the device. Furthermore access to a UNIX shell on the device is possible. Commands can be executed on the command line using the program `adb`.

### 3.4.1 Query devices

ADB allows to query for connected devices. To retrieve a list of connected devices the command `adb devices` has to be executed. This command creates a serial number for each connected device which allows to identify the device when issuing commands. The output of `adb devices` command lists the connected devices by showing their serial number and connection status as can be seen in listing 3.2.

---

```
List of devices attached
323048cb61c870dd      device
emulator-5556        device
emulator-5554        device
```

---

Listing 3.2: Output of the `adb devices` command. Lists the serial number and connection status of all connected devices.

To retrieve more information about the device or emulator the command can be executed using the `-l` flag. The output of `adb devices -l` command lists the connected devices showing their serial number and additional information in order to distinguish between devices as can be seen in listing 3.3. The command can be used to create a list of available devices from the output.

---

```
List of devices attached
323048cb61c870dd      device product:m0xx
                      model:GT_I9300 device:m0 transport_id:5
```

---

### 3 Android Build System

```
emulator-5556          device product: sdk_gphone_x86
  model: Android_SDK_built_for_x86 device: generic_x86
  transport_id: 3
emulator-5554          device product: sdk_gphone_x86
  model: Android_SDK_built_for_x86 device: generic_x86
  transport_id: 1
```

---

Listing 3.3: Output of the `adb devices -l` command. Lists the serial number of all connected devices with additional information to be able to distinguish between devices.

#### 3.4.2 ADB Commands

This section explains the use of ADB commands, besides the `query devices` command in section 3.4.1, which are used by the implemented solution in chapter 4. They are used to automate the mutation testing process. This section does not list all available commands. When issuing a command while multiple devices are connected, the target device must be defined. To specify the target device the serial number of the device has to be passed to the command using the `-s` flag. The serial number can be taken from the output of the `query devices` command as shown in listing 3.2.

The following sections 3.4.2.1 - 3.4.2.6 explain the *install* command to install an APK file on a device, the *uninstall* command to remove a package from a device, the *pull* command to copy files and directories from the device to a local location and the *shell* command using the Activity Manager *am* and Package Manager *pm* tools.

##### 3.4.2.1 Shell

The ADB `shell` command can be used to open a UNIX shell on the device to issue commands or to directly run commands without opening a shell. For example the command `adb shell rm <path-to-file>` allows to delete a file on the device. The Activity Manager tool *am* on the device allows to execute system actions such as start a service, start an activity or start a test runner for instrumented tests. To start instrumented tests see section 3.4.2.6.

The Package Manager tool *pm* can be used to perform actions on installed packages such as uninstall or clear data.

### 3.4.2.2 Install

The ADB `install` command allows to install or reinstall an existing installation of an APK file. To simply install an APK the following command has to be run `adb install <apk-file>`, where `<apk-file>` is the path to the APK file. By setting the `-r` option an existing installation is replaced but the data is kept. To allow installation of an instrumented test APK file for running tests on the device the `-t` option has to be set. This command can also be executed using the *am* tool.

### 3.4.2.3 Uninstall

The *pm* command to uninstall a package from a device takes the application package as an argument and additionally removes the data of the package. The command is invoked using the *shell*. To run the command the application package which is normally the application ID has to be used. `adb shell pm uninstall <package>`. See section [3.2.2.4](#) for the application ID.

### 3.4.2.4 Clear

The *pm* command *clear* can be used to remove the data for a package from the system without the need to uninstall the package. As the uninstall command the application package has to be supplied to the clear command. The command is invoked using the *shell*. The command can be used to clear all package data after a test run.

### 3.4.2.5 Pull

The ADB `pull` command allows to copy files and whole directories from the device to a location on the local machine. For example the command

## 3 Android Build System

can be used to copy files after a test execution for further processing on a local machine.

### 3.4.2.6 Run Instrumented Tests

This section describes how to run instrumented tests using the ADB shell command from section 3.4.2.1 to execute the `am instrument` command. To run the tests the application and test APK files have to be installed on the emulator or mobile device first (see section 3.4.2.2 for installation with ADB). Subsequently the `am instrument` must be executed for the test package, see listing 3.4 for the general syntax of the command. The command has the ability to control which tests should be run.

---

```
adb shell am instrument [flags] <test-package>/<runner>
```

---

Listing 3.4: General syntax of the `am instrument` command to run tests from a test package. `<test-package>` is the package name of the test application and `<runner>` the class of the Android test runner that should be used.

The arguments `<test-package>` and `<runner>` in listing 3.4 are the package name of the test application and the class of the Android test runner that should be used to run the tests such as *android.support.test.runner.AndroidJUnitRunner*. *AndroidJUnitRunner* can run JUnit3 or JUnit4 test classes as well as test classes that use test frameworks like Espresso or UI Automator (see section 2.3).

Important flags for `am instrument` are `-w` that forces the shell to keep open and wait until the tests finish and the `-e` flag to provide test options as key value-pairs to the specified instrumentation runner.

## 4 Pimutdroid Gradle Plugin

This chapter describes the implemented solution to perform mutation testing for an Android project. In section 4.1 a general overview of the plugin is described, section 4.2.1 displays the implemented mutation process and important concepts used by the plugin. The configuration of the developed plugin is described in section 4.3. Finally, section 4.4 provides an overview of the available reports concerning the mutation analysis which are created by the plugin. In chapter 5 a more detailed view on the implementation on the plugin is given.

### 4.1 Plugin Overview

In order to perform mutation testing for an Android application that is easy to use and easy to integrate into the current Android development environment a Gradle plugin named *Pimutdroid* has been developed. Section 4.2 provides an overview of the implemented analysis process and important concepts, as well as the configuration of the plugin. The developed plugin serves as a bridge between local Android unit tests and tests that are run on an emulator or a real device. The plugin uses the Android SDK Platform Tools and Android Gradle Build Tools, which were explained in chapter 3, to build the APK files for the application under test and enables mutation testings for local unit tests as well as instrumented unit tests and UI tests.

Pimutdroid uses Pitest mutation framework, also known as PIT, to mutate the code base of the project and to perform mutation testing for local unit tests (Coles, 2018c). To integrate Pitest into the Gradle build system, the developed plugin uses the in section 2.5 mentioned Gradle integration for Pitest by Wrotniak (2018). The plugin allows to run the Pitest mutation

## 4 Pimutdroid Gradle Plugin

analysis for local unit tests on Android projects. This plugin is an augmented version of the Pitest Gradle integration plugin for traditional Java projects by Zajączkowski (2018). To export the mutated files from Pitest, a Pitest plugin by Coles (2018d) is used, because Pimutdroid uses an Pitest version which does not provide this functionality by default.

For every mutant created by Pitest, Pimutdroid builds an APK file and tests the APK against chosen tests from the test APK on an emulator or a real device. A user can configure the developed plugin via Gradle DSL and for example define which classes the user wants to mutate, which mutators should be used and which tests the user wants to run the mutants against (see section 4.3). After running the tests against the mutants, an Extensible Markup Language (XML) report is created which includes a general overview and a detailed view on the mutation outcome as well as the overall mutation score of the test set 4.4.

## 4.2 Mutation Analysis Tool

Pimutdroid is a Gradle plugin which is written in Java (2018b) and Groovy (2018), it uses the Gradle public API to provide tasks for mutation testing which are added in addition to Android Gradle plugin tasks to the development environment. The user can run these tasks independently to perform different steps of the mutation analysis process for the Android application. The steps of the mutation analysis done by Pimutdroid are described in section 4.2.1. Additionally in sections 4.2.2, 4.2.3 and 4.2.4 important concepts and terms are explained for the implementation details in chapter 5.

### 4.2.1 Mutation Analysis Process

This section describes the process of how Pimutdroid runs mutation analysis on an Android project. Of the following steps which are performed by the plugin in order to generate the mutation result, steps 2 to 8 can be executed by the user as a Gradle task, step 1 is the configuration of the plugin using Gradle DSL:



## 4.2 Mutation Analysis Tool

1. First the user decides which classes should be mutated and configures the used mutation operators and defines how many mutants per class should be created maximally. The configuration of the plugin is described in section 4.3. The mutation operators the user can choose from are explained in section 4.2.4.
2. Next the plugin assembles the application APK of the current project and assembles the test APK against which all mutant APK files and the original application created by the plugin are tested. To create the APK files, Pimutdroid uses the Android Gradle assemble tasks and performs following steps:
  - a) Assemble application APK using assemble task for the configured product flavor and build type.
  - b) Assemble test APK using assemble task for *androidTest* for the configured product flavor and build type.
  - c) Store assembled application APK and test APK files for later use.
  - d) Backup compiled application class files. The backup is used to restore the correct byte code in the mutation phase after a mutant was created.
3. Using Pitest, the application source code for the configured build variant is compiled and the byte code of the defined classes are mutated. The mutated class files and additional metadata created by the Pitest export plugin are stored for later use. They are used when the mutant APK files are assembled and the Pimutdroid report is created. The following steps are performed:
  - a) Mutate byte code by running the Gradle Pitest task for the configured product flavor and build type.
  - b) Run mutation analysis on local unit tests and create the Pitest mutation report. Reports in Hypertext Markup Language format and XML format are created.
  - c) Store the mutated class files and metadata files by Pitest for later use.

## 4 Pimutdroid Gradle Plugin

4. Create mutation metadata for the mutated class files. The plugin assigns an identifier to every mutant called MUID (see *Mutant Identifier* section 4.2.2) and creates a file containing information about the mutant (see section 4.2.3). The data stored for each mutant consist of its MUID, mutated class data, the mutation operator used and whether the mutant was already killed by the local unit tests. This metadata is later used in the creation of the mutation report and in the build process of the mutants.
5. For each mutant an APK file is built using the Android Gradle assemble task for the configured product flavor and build type. Each APK file is stored in combination with its metadata file under the mutant output directory. Additionally, a log file of each build is stored in a separate build log directory.
6. Next the unmutated original application APK as well as the test APK are installed onto an emulator or a real device. The application is tested against all configured tests and the generated result is stored as the expected result for comparison with mutant test results.
7. Each mutant APK file is then installed onto an emulator or a real device and the tests are executed. The mutant test result is pulled from the device and is stored as the actual test result under the directory containing the mutant APK and its metadata.
8. Each actual mutant result is compared with the expected result to determine whether the mutant was killed by the test set or not. If the results differ, the mutant is marked as killed. The plugin gathers the outcome for each mutant and creates a mutation result report in XML format under the report directory and prints the mutation score to the console. Equivalent mutants are not detected by the process.

The Pimutdroid mutation process is inspired by the proposed proof-of-concept tool by L. Deng, Mirzaei, et al. (2015) in section 2.5.1. They differ from each other as the developed plugin does not support XML mutation operators and does not swap out layout files or the Android manifest file in order to create mutants. All steps in the developed plugin can be executed

## 4.2 Mutation Analysis Tool

using the Gradle wrapper from the command line (see section 3.1) or from the Android Studio IDE using the Android Gradle plugin (see section 3.2). The mutants can be executed in parallel on configured devices or on all connected devices. By default Pimutdroid uses the device defined in the Java system variable `ANDROID_SERIAL`. This behaviour is inspired by the Android Gradle plugin by Google (2018d), listing 4.1 display the use of the variable by the Android Gradle plugin to pass one or more serial numbers of devices to the plugin. If the system variable is not set, the first device found will be used instead.

### 4.2.2 Mutation Identifier - MUID

An unique identifier is assigned to each mutant created by Pimutdroid. This mutation identifier is called *MUID*. The MUID is used to identify a mutant by the plugin and to find the associated files needed for mutation analysis. Furthermore it is used as the name of a file referred to as *markerfile*, which stores the metadata of the mutant in XML format. The markerfile will be explained in the following section 4.2.3. Using the MUID it is easier for a user to find the mutant and the data stored with it. Additionally the MUID is used by the build task of the plugin which will be explained in section 5.2. Pitest exports the generated mutants and assigns to each mutant an arbitrary consecutive number per single class it mutated (Coles, 2018d). Pimutdroid creates the MUID from the full class name of the mutant and the number assigned from Pitest on export, referred to as sub identifier. The structure of the MUID is as follows:

`<full class name>_<number of PIT export>`

For example the following MUID is the identifier of the first mutant of a class named "DisplayMessageActivity" from package "at.woodstick.mysampleapp", the sub identifier "0" is the assigned number from the Pitest export:

`at.woodstick.mysampleapp.DisplayMessageActivity_0`

## 4 Pimutdroid Gradle Plugin

---

```
...
IDevice[] devices = bridge.getDevices();

if (devices.length == 0) {
    throw new DeviceException("No connected devices!");
}

final String androidSerialsEnv =
    System.getenv("ANDROID_SERIAL");
final boolean isValidSerial = androidSerialsEnv !=
    null && !androidSerialsEnv.isEmpty();

final Set<String> serials;
if (isValidSerial) {
    serials =
        Sets.newHashSet(Splitter.on(',').split(androidSerialsEnv));
} else {
    serials = Collections.emptySet();
}
...
```

---

Listing 4.1: Shows a code snippet from the class `ConnectedDeviceProvider` of `com.android.tools.build/builder` JAR file in version 3.0.1 of the Android Gradle plugin by Google (2018d). It displays how the system variable `ANDROID_SERIAL` is used to pass the serial number of a device or comma separated serial numbers which should be used by the plugin. Pimutdroid uses a similar approach but only supports passing one serial number by the system variable `ANDROID_SERIAL`.

## 4.2 Mutation Analysis Tool

For the application packages and classes, Pitest creates a HTML report which displays the line coverage value and the according mutation value for local unit tests. The MUID can be used to locate the referenced mutant and identify its mutation in the source code. The Pitest report is described in section 4.4.

### 4.2.3 Mutant Markerfile

For each mutant, the plugin creates a file called *markerfile* which stores the metadata of the mutant. The name of the file is the MUID of the mutant as described in section 4.2.2. The file uses the filename extension “.muid”.

The structure of the name is the following:

```
<full class name>_<number of PIT export>.<filename extension>
```

Using the example MUID at `.woodstick.mysampleapp.DisplayMessageActivity_0` from section 4.2.2, the name of the corresponding markerfile is:

```
at.woodstick.mysampleapp.DisplayMessageActivity_0.muid
```

The metadata of a mutant is stored in XML format as can be seen in listing 4.2. The data the markerfile contains is composed of the following information:

- `<muid>`: Mutation Identifier of the mutant.
- `<clazzPackage>`: Package name of the mutated java class.
- `<clazzName>`: Class name of the mutated java class.
- `<clazz>`: The full class name consisting of package and class name.
- `<method>`: The name of the affected method where the mutation was applied.
- `<mutator>`: The full class name of the applied mutation operator also called mutator.

## 4 Pimutdroid Gradle Plugin

- `<filename>`: The filename the java class resides in.
- `<lineNumber>`: The linenumber in source the mutation was applied at<sup>1</sup>.
- `<description>`: A description of the applied mutation.
- `<indexes>`: First index of the indexes field which contain the instructions within the method a mutation (Coles, 2018b).
- `<killedByUnitTest>`: Indicates whether the mutant is killed by the local unit tests or not, after mutation analysis by Pitest.

The markerfile is stored under the same location as the mutated class file it is associated to. It is created from a file called “details.txt” that contains the serialized instance of the Pitest class `MutantDetails` (Coles, 2018a). Additionally the XML report created by Pitest is used to find out whether a mutant was already killed by the local unit test or not.

---

<sup>1</sup>Only as a reference because Pitest does not mutate the source code directly

---

```

<MutantDetails>
  <muid>
    at.woodstick.mysampleapp.DisplayMessageActivity_0.muid
  </muid>
  <clazzPackage>at.woodstick.mysampleapp</clazzPackage>
  <clazzName>DisplayMessageActivity</clazzName>
  <clazz>
    at.woodstick.mysampleapp.DisplayMessageActivity
  </clazz>
  <method>onCreate</method>
  <mutator>
    org.pitest.mutationtest.engine.gregor
    .mutators.VoidMethodCallMutator
  </mutator>
  <filename>DisplayMessageActivity.java</filename>
  <lineNumber>12</lineNumber>
  <description>
    removed call to
      android/support/v7/app/CompatActivity::onCreate
  </description>
  <indexes>5</indexes>
  <killedByUnitTest>>false</killedByUnitTest>
</MutantDetails>

```

---

Listing 4.2: Content of a mutant markerfile created by Pimutdroid for a mutant of a class named `DisplayMessageActivity`. It contains metadata of a mutant like its MUID, the applied mutation operator, what code part got mutated, what class was affected and whether the mutant was killed by the local unit tests or not.

In the following section 4.2.4 the mutators available in PIT will be explained. These mutation operators are used by Pimutdroid and can be configured by the user.

#### 4.2.4 Mutators

The following sections describe the non-experimental mutators or mutation operators available in Pitest (Coles and Penndorf, 2018g). Pimutdroid can be configured to use a set of these operators for the mutation analysis.

## 4 Pimutdroid Gradle Plugin

The configuration of the plugin is explained in section 4.3. The mutations of Pitest are performed on the byte code rather than on the source code, which is generally faster than source code mutation (Coles, 2018c; Coles and Penndorf, 2018g; Jia and Harman, 2011). The mutants created with the configured mutation operators are used for local unit tests, instrumented unit tests and UI tests. The following subsections (4.2.4.1) - (4.2.4.11) are based on the work of Coles and Penndorf (2018g) and briefly explain the mutators. For each operator the key which is used to configure the mutation operator is listed.

### 4.2.4.1 Conditionals Boundary Mutator

This mutator is activated per default and can be configured individually using the key `CONDITIONALS_BOUNDARY`. It replaces a relational operator like `<`, `<=`, `>`, `>=` with its boundary counterpart (Coles and Penndorf, 2018a). This is used to seed mutations that simulate for example an *Off-by-one-Error*. This mutator is a subset of the ROR operator explained in section 2.1.3. The negate conditionals mutator and remove conditionals in sections 4.2.4.5 and 4.2.4.10 likewise belong to the set of the ROR operator. The operator does not mutate every operand with all operands which results in the creation of lesser redundant mutants (R. Just, Kapfhammer, and Schweiggert, 2012; Kaminski, Ammann, and Jeff Offutt, 2011).

For “lesser” and “lesser than” this means:

`<` will be mutated to `<=`

and

`<=` will be mutated to `<`

For “greater” and “greater than” this means:

`>` will be mutated to `>=`

and

`>=` will be mutated to `>`



### 4.2.4.2 Increments Mutator

This mutator is activated by default and can be configured individually using the key INCREMENT. It replaces increments, decrements, assignment increments and assignment decrements with its counterpart. The mutation operator is only applied to *local* variables (Coles and Penndorf, 2018c).

If increments on member variables should also be mutated, the Math Mutator in section 4.2.4.4 must be applied in combination with the Increments Mutator.

### 4.2.4.3 Invert Negatives Mutator

This mutator is activated by default and can be configured individually using the key INVERT\_NEGS. It removes the negation of integer or floating point numbers (Coles and Penndorf, 2018e).

### 4.2.4.4 Math Mutator

This mutator is activated by default and can be configured individually using the key MATH. An integer or floating-point operation is replaced with an operation selected from a mapping table which is shown in table 4.1. The mutator does not mutate the +-operator for strings, but it mutates increments, decrements, assignment increments and assignment decrements on member variables (Coles and Penndorf, 2018f).

The Increments Mutator in section 4.2.4.2 only applies to local variables, if both local and member variable increments should be mutated, both operators Increments Mutator and Math Mutator must be active in the mutation analysis.

## 4 Pimutdroid Gradle Plugin

Original	Mutated
+	-
-	+
*	/
/	*
%	*
&	
	&
^	&
<<	>>
>>	<<
>>>	<<<

Table 4.1: Binary arithmetic operations mapping table used by the Math Mutator. Note. Reprinted from (Coles and Penndorf, 2018f).

### 4.2.4.5 Negate Conditionals Mutator

This mutator is activated by default and can be configured individually using the key `NEGATE_CONDITIONALS`. It will negate conditionals like `==`, `!=`, `<=`, `>=`, `<`, `>` and replace them according to a mapping table as shown in table 4.2. In comparison to the Conditionals Boundary Mutator in section 4.2.4.1 a mutation done by the Negate Conditionals Mutator is easier to detect for a test suite, hence it is less stable (Coles and Penndorf, 2018h; Kaminski, Ammann, and Jeff Offutt, 2011). As the negate conditionals mutator and remove conditionals in sections 4.2.4.5 and 4.2.4.10, this mutator is a subset of the ROR operator explained in section 2.1.3.

### 4.2.4.6 Return Values Mutator

This mutator is activated by default and can be configured individually using the key `RETURN_VALS`. It replaces the return value of a method depending on the return type of the affected method (Coles and Penndorf, 2018k).

Original	Mutated
==	!=
!=	==
<=	>
>=	<
<	>=
>	<=

Table 4.2: Conditional negation mapping table used by the Negate Conditionals Mutator. Note. Reprinted from (Coles and Penndorf, 2018h).

For a return value of type boolean the unmutated value true will be replaced with false and vice versa. The unmutated return value x of type int, byte or short will be replaced with zero. With the exception that if value x is zero it will be replaced with one. A return value of type long will be replaced with its incremented value. The unmutated value of an object will be replaced with null if the value is a non-null value, otherwise a Java RuntimeException will be thrown instead of the null return value. For floating-point numbers of type float or double the unmutated return value x which is not NAN will be replaced with  $-(x+1.0)$ . A value of NAN will be replaced with zero (Coles and Penndorf, 2018k).

#### 4.2.4.7 Void Method Call Mutator

This mutator is activated by default and can be configured individually using the key VOID\_METHOD\_CALLS. It removes calls to void methods, exceptions in this regard are constructors of classes, these calls will not be removed (Coles and Penndorf, 2018l). To mutate calls to constructors the Constructor Call Mutator in section 4.2.4.8 has to be used. To mutate non void method calls see Non Void Method Call Mutator in section 4.2.4.9.

This mutator tends to create trivial mutants for Android classes if the mutated class is a type of an Android activity, as shown in an example in section 2.1.4. If the class implements a life cycle it should always call the superclass method (Google, 2018a). These methods are void methods and

## 4 Pimutdroid Gradle Plugin

---

```
E/AndroidRuntime: FATAL EXCEPTION: main
Process: at.woodstick.mysampleapplication, PID: 28803
android.util.SuperNotCalledException: Activity
{at.woodstick.mysampleapplication/
    at.woodstick.mysampleapplication.MainActivity}
    did not call through to super.onStart\(\)
    at android.app.Activity.performStart(Activity.java:6698)
    at android.app.ActivityThread.
        performLaunchActivity(ActivityThread.java:2628)
    ...
```

---

Listing 4.3: Shows the fatal android exception which is thrown when an activity does not call its superclass life cycle method when implementing it. The Void Method Call mutator in section 4.2.4.7 tends to create trivial mutants for activity classes when remove these void life cycle methods.

when removed by the mutator the application will crash on startup or when the application is paused or stopped and a fatal exception is thrown as shown in listing 4.3. The life cycle methods `onStart`, `onRestart`, `onResume`, `onPause` and `onStop` are affecting the tests, the only exception is `onDestroy` as it does not crash the application.

### 4.2.4.8 Constructor Call Mutator

This mutator is not activated by default and can be activated using the key `CONSTRUCTOR_CALLS` in the configuration. It removes calls to constructors with `new` and replaces them with `null` value (Coles and Penndorf, 2018b).

### 4.2.4.9 Non Void Method Call Mutator

This mutator is not activated by default and can be activated using the key `NON_VOID_METHOD_CALLS` in the configuration. It replaces calls to non void methods with the default value of the Java type of the methods return value. This does not affect constructor or void method calls (Coles and Penndorf, 2018i).

For constructor mutation see previous section [4.2.4.8](#) and for void method calls see section [4.2.4.7](#).

### 4.2.4.10 Remove Conditionals Mutator

This mutator is not activated by default and can be activated using the key `REMOVE_CONDITIONALS` in the configuration. The mutator replaces whole conditional statements with a boolean `true` or `false` to ensure that either the `if` or `else` block is executed (Coles and Penndorf, 2018j).

`REMOVE_CONDITIONALS` activates the four following specialized versions which can be configured independently of one another (Coles and Penndorf, 2018j):

- `REMOVE_CONDITIONALS_EQ_IF`: Force equality check (`==`, `!=`) to execute `if` block, expression is replaced with `true`
- `REMOVE_CONDITIONALS_EQ_ELSE`: Force equality check (`==`, `!=`) to execute `else` block, expression is replaced with `false`
- `REMOVE_CONDITIONALS_ORD_IF`: Force order check (for example: `<=` or `>`) to execute `if` block, expression is replaced with `true`
- `REMOVE_CONDITIONALS_ORD_ELSE`: Force order check (for example: `<=` or `>`) to execute `else` block, expression is replaced with `false`

### 4.2.4.11 Inline Constant Mutator

This mutator is not activated by default and can be activated using the key `INLINE_CONSTS` in the configuration. It replaces the inline constant value of non-final variables (Coles and Penndorf, 2018d).

## 4.3 Plugin Configuration

Section [4.3.1](#) describes how the plugin can be installed in order to perform mutation testing for an Android application. Furthermore, important con-

## 4 Pimutdroid Gradle Plugin

figuration properties are explained in section 4.3.2. Pimutdroid supports the configuration of all input and output directories which are used by the plugin. It is possible to enable parallel test execution on multiple emulators by the configuration as well as definition of the classes which should be mutated and which tests should be run.

### 4.3.1 Install Plugin

The developed Gradle plugin is a binary plugin as explained in section 3.1 and available as a JAR file which can be added to a Gradle project. Therefore the plugin is added to the classpath configuration of the build script of the root project as shown in listing 4.4. The dependencies of the plugin have to be on the classpath as well, if supplying the JAR file from a local directory the three dependencies have to be set in the build script as can be seen in listing 4.4. The plugin uses the library Jackson (2018) to serialize and deserialize XML files, the Pitest Android Gradle plugin by Wrotniak (2018) to integrate Pitest and the Pitest export plugin by Coles (2018d) to have access to the mutated class files. Alternatively the JAR of Pimutdroid and the export plugin could be installed into an artifact repository such as Maven<sup>2</sup>, then the dependencies would be resolved by Gradle and the build script configuration only needs the definition for Pimutdroid and the Pitest export plugin, as can be seen in listing 4.5.

When the plugin and its dependencies are available to the build script the plugin can be integrated into the project. As shown in listing 4.6 Pimutdroid is applied using the “apply” method of Gradle. The ID of the plugin is “at.woodstick.pimutdroid”. The code in listing 4.6 suffices to add the tasks for mutation testing using default configurations. Pimutdroid offers an extension closure named “pimut” which can be used to adopt default values, additionally Pitest can also be configured using this closure.

---

<sup>2</sup> <https://maven.apache.org/>

## 4.3 Plugin Configuration

---

```
buildscript {
    repositories {
        jcenter()
        google()
    }

    configurations.maybeCreate("pitest")

    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.1'
        classpath 'com.dicedmelon.gradle:jacoco-android:0.1.2'

        // Add pimutdroid mutation testing dependencies
        classpath group: "com.fasterxml.jackson.dataformat",
            name: "jackson-dataformat-xml", version: "2.9.2"
        classpath group: "pl.droidsonroids.gradle", name:
            "gradle-pitest-plugin", version: "0.1.5"
        classpath
            files("mutationLibs/gradle-pimutdroid-plugin-0.0.1.jar")
        pitest
            files("mutationLibs/pitest-export-plugin-0.1-SNAPSHOT.jar")
    }
}
```

---

Listing 4.4: Shows the buildscript block of the root project build.gradle file of the Paintroid project. Pimutdroid and its dependencies are added to the build script in order to be able to apply the plugin to the project. The binary plugin is applied from a local directory "mutationLibs".

## 4 Pimutdroid Gradle Plugin

---

```
buildscript {
    repositories {
        jcenter()
        google()
        mavenLocal()
    }

    configurations.maybeCreate("pitest")

    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.1'
        classpath 'com.dicedmelon.gradle:jacoco-android:0.1.2'

        // Add pimutdroid mutation testing dependencies
        classpath group: "at.woodstick", name:
            "gradle-pimutdroid-plugin", version: "0.0.1"
        pitest group: "org.pitest.plugins", name:
            "pitest-export-plugin", version: "0.1-SNAPSHOT"
    }
}
```

---

Listing 4.5: Shows the buildscript block of the root project build.gradle file of the Paintroid project. Pimutdroid and its dependencies are added to the build script in order to be able to apply the plugin to the project. The binary plugin is applied from a the local maven repository using the repository “mavenLocal()”.

---

```
apply plugin: "at.woodstick.pimutdroid"
```

---

Listing 4.6: Shows how Pimutdroid is applied to a build script using the apply method of Gradle. In order to install the plugin the plugin ID “at.woodstick.pimutdroid” is needed to reference the main plugin class to load.



### 4.3.2 Configure Plugin

Pimutdroid offers an extension object named “pimut” which is associated with the project the plugin has been applied to. The plugin wraps the Pitest Gradle integration and exposes an extension property “pitest” which enables to configure the default values of Pitest. If no additional configuration to the default configuration is supplied, then using the default mutation operators of Pitest from section 4.2.4 are used to create the mutants for all classes under the main Java package. By default all tests of the test APK are executed against the original and mutated APK files, if a mutant was killed by a local unit test it will still be tested against the tests which have to run on an emulator or a real device. The plugin executes the mutants consecutively on a single emulator or device.

Pimutdroid gives the possibility to exclude mutants that were already killed by a local unit test from test execution against the instrumented unit tests and UI tests. This approach is inspired by the “do fewer” mutation testing approaches from section 2.1.4. This tackles the problem that an enormous amount of mutants has to be assembled as APK files and that each has to run against the tests on an emulator or a real device. In the resulting report the mutant is still listed and marked as killed, so the property named “ignoreKilledByUnitTest” speeds up mutation testing for Android applications if local unit tests which already kill mutants exist.

Another approach to tackle the execution cost problem from section 2.1.4 is to run the tests in parallel. This “do smarter” approach can be configured using the “devices” extension property as can be seen in listing 4.7. As explained in section 4.2.1 by default Pimutdroid is reading the serial number of the device on which the tests should be executed from the Java system variable named ANDROID\_SERIAL. This feature can be ignored setting the property “ignoreAndroidSerial”. Now the first available device returned by the ADB query devices command from section 3.4.1 will be used. When the property “parallelExecution” is enabled, all available devices will be used to execute the tests in parallel. If parallel execution is enabled and the system variable is ignored, the user can define the devices which should be used by defining their serial numbers in the “serialNumbers” list property. If this property is not empty, the plugin favors this serial numbers, however it is

## 4 Pimutdroid Gradle Plugin

---

```
pimut {
    devices {
        serialNumbers = []
        parallelExecution = true
        ignoreAndroidSerial = true
    }
}
```

---

Listing 4.7: Shows the configuration for parallel execution of the tests on all available devices. This “do smarter” approach from section 2.1.4 can be to tackle the execution cost problem.

checked on execution if the device is available or not. Unavailable devices defined in “serialNumbers” will be ignored by the plugin.

The defined mutation operators control which and how many mutants are created by Pitest. By default the Pitest default mutation operators are used, however they can be customized using the extension property “pitest.mutators”. Listing 4.8 shows the configuration of the increments, return values, negate conditionals, invert negatives and conditionals boundary operators using their respective keys from section 4.2.4. Pitest gives the possibility to set a maximum number of mutants which should be created per class using the “pitest.maxMutationsPerClass” property as shown in listing 4.8. By default, the value is set to zero which means that there is no restriction in place.

Using the “instrumentationTestOptions” configure closure the target classes for mutation as well as the target instrumented tests can be configured globally. The closure takes a list of package or full class globs to for example target a subset of the main package. Using the “!” character whole packages or classes can be excluded from the resulting set of globs. At the moment the glob to exclude mutants only affects the build or test execution task provided by Pimutdroid and not the mutation analysis of the local unit tests and export of mutant files by Pitest. Listing 4.9 displays an example configuration using the glob and exclude glob to target mutants under the “command” and its sub packages, except the “implementation” package. By default all tests of the assembled test APK file will be run, using the property “targetTests” these tests can be restricted. This configuration is

---

```
pimut {
    pitest {
        mutators = [
            "INCREMENTS",
            "RETURN_VALS",
            "NEGATE_CONDITIONALS",
            "INVERT_NEGS",
            "CONDITIONALS_BOUNDARY",
        ]
        maxMutationsPerClass = 20
    }
}
```

---

Listing 4.8: Shows the configuration of the mutation operators from section 4.2.4 as well as the restriction to create a maximum of 20 mutants per class by setting a value to the “maxMutationsPerClass” property.

used in conjunction with AndroidJUnitRunner instrumentation as described in section 3.4.2.6 which supports execution options to run specific tests (Google, 2010). At the moment the supported options for test restriction are:

- Run all tests of the test APK.
- Run a single test class.
- Run multiple test classes.
- Run all tests under a Java package.
- Test timeout in milliseconds to prevent infinite loops (by default 10 seconds).

In listing 4.9 the “targetTests” property is configured to run all tests under a Java package. The “packages” property of “targetTests” closure supports only one package without “\*” characters. The “classes” list property can be used to run a single test class or multiple test classes. If both extension fields are configured the “packages” property is favored by Pimutdroid. Furthermore the test timeout is set to 20 seconds to prevent that mutants which cause an infinite loop are stalling the test execution.

Finally an important configuration provided by Pimutdroid is the definition of build configurations. They can be used to define subsets of the targeted

## 4 Pimutdroid Gradle Plugin

---

```
pimut {
    instrumentationTestOptions {
        targetMutants = [
            "org.catrobat.paintroid.command.*",
            "!org.catrobat.paintroid.command.implementation.*",
        ]

        targetTests {
            packages = [
                "org.catrobat.paintroid.test.junit.command"
            ]
            classes = []
        }
    }
    testTimeout = 20000
}
```

---

Listing 4.9: Shows a configuration of target mutants for classes under the “command” and its sub packages, except the “implementation” package. Additionally instrumented tests under a single package and its sub packages are configured. The “packages” property of “targetTests” closure supports only one package without “\*” characters. In the “classes” list multiple test classes can be configured which should be executed. Furthermore the test timeout is set to 20 seconds to prevent against a mutant which causes an infinite loop.

mutants and allow to independently define target mutants in combination with the mutation operators and the maximum of mutants which should be created. Each build configuration then adds its custom tasks under the name of the configuration which can be chosen freely. The build configurations can be used to achieve that the mutant APK files are created in parallel, as per default they are created subsequently. By using build configurations they can be build and tested in parallel. To achieve this the project has to be cloned for every build configuration which should run in parallel and the task for the specific configuration has to be executed. This tackles the problem that for each mutant the APK file has to be assembled and is in line with the “do smarter” approach from section 2.1.4. Alternatively they can be used to analyze the test set for different mutation operators without the need to change the global definition of the operators under

“pitest.mutators”. Listing 4.10 displays an example configuration where the “buildConfiguration” closure is used to define four build configurations named “commandsMath”, “standard”, “math” and “voidCalls”. The “commandsMath” definition shows how a subset of the classes are configured to be mutated only by the math mutation operator of section 4.2.4.4. The configurations “standard”, “math” and “voidCalls” can be used to analyze the test set against different mutation operators, if desired, also in parallel.

## 4.4 Plugin Report

This section describes how the mutation analysis XML report is created by Pimutdroid. Furthermore the HTML report for local unit tests of Pitest is explained. In order to generate the mutation result the build process steps 1. to 7. explained in section 4.2.1 have to be executed first. To run all steps including the creation of the report, only the Gradle task “generateMutationResult” has to be executed. This task will trigger several tasks, one is the Pitest task to analyze local unit tests and export the mutant class files. Another task assembles the original application and test APK files and creates the expected result file. Additionally tasks prepare the metadata files for the mutants, build the mutants and run them on the configured devices against the test set. Finally, each mutant result file is compared with the expected result file and the report file is written to the result output directory. By default this is “<buildDir>/reports/mutation”, however it can be configured using the plugin extension explained in section 4.3. The default name of the report file is “mutation-result-yyyyMMdd-hhmmss.xml” where “yyyy” is the year, “MM” the number of the month, “dd” they day of the month and “hhmmss” the time it was created. Pimutdroid offers a task “generateMutationResultOnly” which creates the report and only compares the mutant result files with the expected result files, this task does not trigger any other task.

The Pitest report is an HTML page which can be opened in a browser to navigate to each mutated class. The report gives an overview of the total mutation result as well as on per package and per class basis. Figure 4.1 shows the project summary of the Pitest report. It shows the total mutation

## 4 Pimutdroid Gradle Plugin

---

```
pimut {
    buildConfiguration {
        commandsMath {
            targetMutants = [
                "org.catrobat.paintroid.command.*",
                "!org.catrobat.paintroid.command.implementation.*",
            ]
            mutators = [
                "MATH"
            ]
        }

        standard {
            targetMutants = [
                "org.catrobat.paintroid.*"
            ]
            mutators = [
                "INCREMENTS",
                "RETURN_VALS",
                "NEGATE_CONDITIONALS",
                "INVERT_NEGS",
                "CONDITIONALS_BOUNDARY",
            ]
        }

        math {
            targetMutants = [
                "org.catrobat.paintroid.*"
            ]
            mutators = [
                "MATH"
            ]
        }

        voidCalls {
            targetMutants = [
                "org.catrobat.paintroid.*"
            ]
            mutators = [
                "VOID_METHOD_CALLS"
            ]
        }
    }
}
```

---

Listing 4.10: Shows four build configurations “commandsMath”, “standard”, “math” and “voidCalls” which can be used to create and test mutants for different mutation operators, a subset of the target classes or use the configuration to assemble the mutant APK files in parallel.

## 4.4 Plugin Report

coverage of the test set and displays the mutation score as a percentage value (see section 2.1). In this example, from the 1460 mutants created, the three local unit tests killed 18 mutants, which results in a mutation score of 1%. It can be seen that there is nearly no test coverage using local unit tests. A mutated class can be inspected in detail by navigating to the class, this opens a view where the mutated lines are highlighted showing what mutation was applied to the line and whether the test set was able to detect the mutant as can be seen in figure 4.2. The lines are colored in green and red color, a darker green color means that the mutated statement was reached by the test set and that the mutant was killed. A darker red indicates that the line was not reached and the mutant lived, which can be used to write a test case or to optimize an existing test case to detect the mutant. Additionally, as shown in figure 4.3, a summary is listed at the end of the class level view. It consists of the applied mutation operators, the mutants and the outcome for each mutant as well as which tests were examined and killed a mutant.

The XML report created by Pimutdroid lists the mutation score for the project including how many mutants were tested and killed by the test set. Additionally the mutation score and the numbers concerning the mutants are displayed for every package and class file, inner class mutants are counted towards the outer class. Furthermore, the test setup is included in the report consisting of the used instrumentation runner class, the targeted mutants and the test package or test classes which were configured (see section 4.3.2). For each class, as can be seen in listing 4.11, the created mutants with their corresponding MUID from section 4.2.2 and their outcome are listed. Additionally the information of the mutation is displayed, which described which operation was applied to create the mutant. The mutants are grouped together into a section concerning the class of the mutant. Inner classes are displayed in their own mutation group. This “mutantGroup” of a class provides an additional overview of the overall mutants, it shows for the group how many mutants were killed and the resulting mutation score.

The provided MUID of a mutant and the line number can be used in conjunction with the Pitest report to take a closer look at the mutant on source code level. All mutants by the test set are included in the report, whether the “ignoreKilledByUnitTest” configuration property from section

## 4 Pimutdroid Gradle Plugin

---

```
<mutantGroup package="org.catrobat.paintroid.command"
  file="UndoRedoManager.java" class="UndoRedoManager"
  mutants="41" killed="0" score="0.000000">
  <mutant
    id="org.catrobat.paintroid.command.UndoRedoManager_0.muid"
    outcome="LIVED">
    <mutation
      method="getInstance"
      line="48" description="negated conditional"
      mutator="org.pitest.mutationtest.engine.gregor.mutators
        .NegateConditionalsMutator"
    />
  </mutant>
  ...
<mutantGroup package="org.catrobat.paintroid.command"
  file="UndoRedoManager.java" class="UndoRedoManager$1"
  mutants="7" killed="0" score="0.000000">
  <mutant
    id="org.catrobat.paintroid.command.UndoRedoManager$1_0.muid"
    outcome="LIVED">
  ...
```

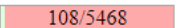
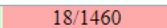
Listing 4.11: Shows the mutant information displayed in the XML report created by Pimutdroid. For each class which was mutated the created mutants and their corresponding information about the mutation are stored. Inner classes are displayed in their own mutation group. The “mutantGroup” provides an overview of how many mutants were created and killed by the test set, leading to the mutation score for the class.

4.3.2 was set or not. When the property is set the report marks this mutants as killed in the report and the plugin does not execute the instrumented tests against these mutants. Thus, the report can give a combined view on local and instrumented test cases.

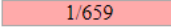
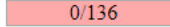
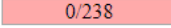
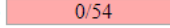
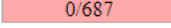
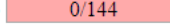
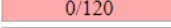
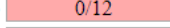
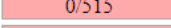
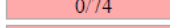
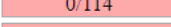
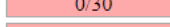
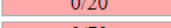
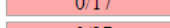
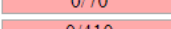
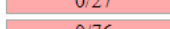
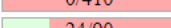
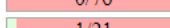
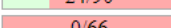
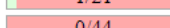
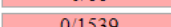
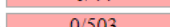
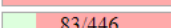
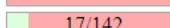
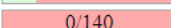
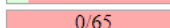
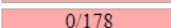
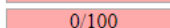
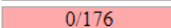
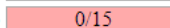




## Pit Test Coverage Report

### Project Summary

Number of Classes	Line Coverage	Mutation Coverage
76	2% 	1% 

### Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
<a href="#">org.catrobat.paintroid</a>	6	0% 	0% 
<a href="#">org.catrobat.paintroid.command</a>	1	0% 	0% 
<a href="#">org.catrobat.paintroid.command.implementation</a>	13	0% 	0% 
<a href="#">org.catrobat.paintroid.dialog</a>	4	0% 	0% 
<a href="#">org.catrobat.paintroid.dialog.colorpicker</a>	7	0% 	0% 
<a href="#">org.catrobat.paintroid.intro</a>	4	0% 	0% 
<a href="#">org.catrobat.paintroid.intro.helper</a>	1	0% 	0% 
<a href="#">org.catrobat.paintroid.iotasks</a>	3	0% 	0% 
<a href="#">org.catrobat.paintroid.listener</a>	5	0% 	0% 
<a href="#">org.catrobat.paintroid.tools</a>	3	27% 	5% 
<a href="#">org.catrobat.paintroid.tools.helper</a>	1	0% 	0% 
<a href="#">org.catrobat.paintroid.tools.implementation</a>	13	0% 	0% 
<a href="#">org.catrobat.paintroid.ui</a>	7	19% 	12% 
<a href="#">org.catrobat.paintroid.ui.button</a>	2	0% 	0% 
<a href="#">org.catrobat.paintroid.ui.dragndrop</a>	3	0% 	0% 
<a href="#">org.catrobat.paintroid.ui.tools</a>	3	0% 	0% 

Report generated by [PIT](#) 1.2.2

Figure 4.1: Shows the Pitest HTML report which gives a summary of the mutation analysis executed on the local unit tests of the Paintroid project. The percentage value mutation coverage is the mutation score for the test set (see section 2.1). It can be seen that there is nearly no test coverage using local unit tests and that the three existing test classes killed 18 of 1460 mutants.

## 4 Pimutdroid Gradle Plugin

```
175
176     public float getScale() {
177 1         return surfaceScale;
178     }
179
180     public synchronized void setScale(float scale) {
181         surfaceScale = Math.max(MIN_SCALE, Math.min(MAX_SCALE, scale));
182     }
183
184     public float getScaleForCenterBitmap() {
185         float ratioDependentScale;
186         float screenSizeRatio = surfaceWidth / surfaceHeight;
187         float bitmapSizeRatio = bitmapWidth / bitmapHeight;
188
189         2         if (screenSizeRatio > bitmapSizeRatio) {
190             ratioDependentScale = surfaceHeight / bitmapHeight;
191         } else {
192             ratioDependentScale = surfaceWidth / bitmapWidth;
193         }
194
195         2         if (ratioDependentScale > 1f) {
196             ratioDependentScale = 1f;
197         }
198     }
```

Figure 4.2: Shows the Pitest HTML report view on class level. It displays the line coverage, shows on which line number a mutation occurred and whether the mutant was killed by the test set, using the darker green color to indicate that the mutation was reached and the mutant was killed. The darker red color indicates that the mutation statement was not reached by the test set and the mutant lived.

## 4.4 Plugin Report

Mutations	
<a href="#">84</a>	1. negated conditional → SURVIVED 2. negated conditional → SURVIVED
<a href="#">86</a>	1. removed negation → NO_COVERAGE
<a href="#">95</a>	1. negated conditional → SURVIVED
<a href="#">109</a>	1. changed conditional boundary → NO_COVERAGE 2. negated conditional → NO_COVERAGE
<a href="#">111</a>	1. changed conditional boundary → NO_COVERAGE 2. removed negation → NO_COVERAGE 3. negated conditional → NO_COVERAGE
<a href="#">112</a>	1. removed negation → NO_COVERAGE
<a href="#">117</a>	1. changed conditional boundary → NO_COVERAGE 2. negated conditional → NO_COVERAGE
<a href="#">119</a>	1. changed conditional boundary → NO_COVERAGE 2. removed negation → NO_COVERAGE 3. negated conditional → NO_COVERAGE
<a href="#">120</a>	1. removed negation → NO_COVERAGE
<a href="#">132</a>	1. mutated return of Object value for org.catrobat.paintroid/ui/Perspective::getCanvasPointFromSurfacePoint t
<a href="#">159</a>	1. mutated return of Object value for org.catrobat.paintroid/ui/Perspective::getSurfacePointFromCanvasPoint t
<a href="#">177</a>	1. replaced return of float value with $-(x + 1)$ for org.catrobat.paintroid/ui/Perspective::getScale → KILLED
<a href="#">190</a>	1. changed conditional boundary → SURVIVED 2. negated conditional → SURVIVED
<a href="#">196</a>	1. changed conditional boundary → SURVIVED 2. negated conditional → SURVIVED
<a href="#">199</a>	1. changed conditional boundary → SURVIVED 2. negated conditional → SURVIVED
<a href="#">203</a>	1. replaced return of float value with $-(x + 1)$ for org.catrobat.paintroid/ui/Perspective::getScaleForCenterB
<a href="#">207</a>	1. replaced return of integer sized value with $(x == 0 ? 1 : 0)$ → NO_COVERAGE
<a href="#">216</a>	1. replaced return of float value with $-(x + 1)$ for org.catrobat.paintroid/ui/Perspective::getSurfaceTranslat
<a href="#">224</a>	1. replaced return of float value with $-(x + 1)$ for org.catrobat.paintroid/ui/Perspective::getSurfaceTranslat

### Active mutators

- INCREMENTS\_MUTATOR
- RETURN\_VALS\_MUTATOR
- NEGATE\_CONDITIONALS\_MUTATOR
- INVERT\_NEGS\_MUTATOR
- CONDITIONALS\_BOUNDARY\_MUTATOR

### Tests examined

- org.catrobat.paintroid.test.PerspectiveTests.testSetScaleAboveMaximum(org.catrobat.paintroid.test.PerspectiveTests) (2 ms)
- org.catrobat.paintroid.test.PerspectiveTests.testInitialize(org.catrobat.paintroid.test.PerspectiveTests) (1188 ms)
- org.catrobat.paintroid.test.PerspectiveTests.testSetScale(org.catrobat.paintroid.test.PerspectiveTests) (1 ms)
- org.catrobat.paintroid.test.PerspectiveTests.testSetScaleBelowMinimum(org.catrobat.paintroid.test.PerspectiveTests) (1 ms)
- org.catrobat.paintroid.test.PerspectiveTests.testMultiplyScaleBelowMinimum(org.catrobat.paintroid.test.PerspectiveTests) (1 ms)
- org.catrobat.paintroid.test.PerspectiveTests.testMultiplyScaleAboveMaximum(org.catrobat.paintroid.test.PerspectiveTests) (1 ms)
- org.catrobat.paintroid.test.PerspectiveTests.testMultiplyScale(org.catrobat.paintroid.test.PerspectiveTests) (10 ms)

Figure 4.3: Shows the Pitest HTML report summary under a class view which lists the mutation applied to the class and which mutants were killed. Furthermore the used mutation operators are shown and which tests were examined and killed a mutant.



# 5 Implementation Details

Chapter 5 describes the basic structure of the implemented plugin in section 5.1. In section 5.2 the process of creating mutant APK files utilizing Gradle and the Android Gradle plugin is displayed. Finally, in section 5.3 test execution for the mutated APK files is described.

## 5.1 Structure

The mutation analysis tool is structured into three separate plugins. A Pimutdroid base plugin, a Pimutdroid Pitest wrapper plugin for local tests and the full Pimutdroid plugin, which is described in chapter 4, to enable mutation testing for instrumented tests.

The base plugin defines constants used throughout the plugin (see listing 5.1). The constants defined by the base plugin are the name of the directory to store the report (1), the command line project property to pass a MUID to a build (2), the default Android test runner class (3), the extension name for plugin configuration (4), the task group of the Gradle tasks (5) and the default Pitest version (6). Furthermore, when the base plugin is applied to the project, the plugin verifies that the Android Gradle plugin is already applied to the project as can be seen in listing 5.2 and applies the Pitest Gradle plugin afterwards, if it is not yet activated. If the Android plugin is not added to the project Pimutdroid ends the Gradle configuration phase with an exception (see section 3.1 for Gradle build phases). When applying the full plugin of Pimutdroid as described in section 4.3, the Pimutdroid Pitest wrapper plugin is additionally applied to the project. The wrapper plugin likewise applies the base plugin to the project to verify that the Android plugin is used. The wrapper plugin supports build

## 5 Implementation Details

---

```
static final String REPORTS_DIR_NAME = "reports"; (1)
public static final String PROPERTY_NAME_MUID =
    "pimut.muid"; (2)
public static final String RUNNER =
    "android.support.test.runner.AndroidJUnitRunner"; (3)
public static final String PLUGIN_EXTENSION = "pimut"; (4)
public static final String PLUGIN_TASK_GROUP = "Mutation";
    (5)
public static final String PITEST_VERSION = "1.2.2"; (6)
```

---

Listing 5.1: The constants predefined by the base plugin are the name of the directory to store the report (1), the command line project property to pass a MUID to a build (2), the default Android test runner class (3), the extension name for plugin configuration (4), the task group of the Gradle tasks (5) and the default Pitest version (6).

configurations, the configuration of Pitest and enables to analyze the local unit tests. Therefore, a task called “mutateClasses” is added to the project which configures and triggers the according task of Pitest. The full plugin uses the complete configuration which is offered by the plugin and adds in total 29 tasks under the task group “mutation” as defined by the base plugin. For each build configuration defined additional nine task are created which are specific to the configuration.

The following 29 tasks are added by the full plugin:

1. **availableDevices:** Helper task which displays the connected devices, uses ADB query devices command from section 3.4.1.
2. **backupApks:** Depends on the assemble tasks for the application APK and test APK and copies the assembled files from the Android output folder to the mutation application directory.
3. **backupCompiledClasses:** Copies the compiled class files of the application to the mutation application directory from where they can be restored later.
4. **buildMutantApks:** For each configured target mutant it starts the build task of the APK file. Triggers the Pitest task and tasks that prepare mutation files.

5. **buildMutantApksOnly**: Same as “buildMutantApks” but does not depend on any task.
6. **buildOnlyMutantApk**: For a given MUID it builds the mutant APK file, this task is executed from “buildMutantsApks”.
7. **cleanActualResultFiles**: Helper task to remove all mutant result files.
8. **cleanMutantAppFiles**: Helper task to clean the mutant application directory (remove backup of class files, APK files and expected result file).
9. **cleanMutantBuildLogFiles**: Helper task which removes the mutant build log files.
10. **cleanMutantClasses**: Helper task which removes the Pitest output directory.
11. **cleanMutantResultFiles**: Helper task which removes all generated Pimutdroid XML reports.
12. **cleanMutation**: Helper task which triggers “cleanMutantClasses”, “cleanMutationOutput” and “cleanMutantClasses” tasks.
13. **cleanMutationOutput**: Helper task to remove complete mutation output folder.
14. **configuredDevices**: Displays the configured available devices, to verify that “devices” is configured properly.
15. **generateExpectedResult**: Installs the origin application APK and test APK file on a device and creates the expected result file.
16. **generateMutationResult**: Executes the whole mutation analysis process from section [4.2.1](#).
17. **generateMutationResultOnly**: Generates the XML report only.
18. **injectMutantAfterCompileByMarkerFile**: Injects the mutated class file into the compiled class files.
19. **mutantClassesList**: Helper task which lists all mutated class files.
20. **mutantMarkerList**: Helper task which lists markerfiles.
21. **mutantXmlResultList**: Helper task which lists mutant result files.

## 5 Implementation Details

22. **mutateClasses**: Triggers Pitest mutation analysis task.
23. **pimutInfo**: Info task of the plugin which displays configuration information.
24. **prepareApplicationMutationData**: Triggers the assemble, backup class files and generated expected result task.
25. **prepareMutationFiles**: Creates MUID and markerfiles for the mutated class files.
26. **restoreCompiledClasses**: Task to restore the origin class files after a mutant was assembled.
27. **testMutants**: Triggers test execution on configured devices of the mutant APK files. Depends on “prepareApplicationMutationData” and “buildMutantApks” tasks.
28. **testMutantsGenerateResultOnly**: Task to only run the test execution and create the XML report.
29. **testMutantsOnly**: Task to only run the test execution. Depends on no other task.

For each build configuration the following nine tasks are added additionally by the full plugin:

1. buildMutantApksOnly<buildConfig>
2. buildMutantApks<buildConfig>
3. generateMutationResultOnly<buildConfig>
4. generateMutationResult<buildConfig>
5. mutateClasses<buildConfig>
6. prepareMutationFiles<buildConfig>
7. testMutantsGenerateResultOnly<buildConfig>
8. testMutantsOnly<buildConfig>
9. testMutants<buildConfig>



```
...
@Override
public void apply(Project project) {
    PluginContainer pluginContainer = project.getPlugins();

    if(!pluginContainer.hasPlugin(AndroidBasePlugin.class)) {
        throw new GradleException(String.format("Android plugin
            must be applied to project"));
    }

    if(!pluginContainer.hasPlugin(PitestPlugin.class)) {
        project.getPluginManager().apply(PitestPlugin.class);
    } else {
        LOGGER.info("pitest plugin already applied.");
    }
}
...

```

---

Listing 5.2: Pimutdroid base plugin verifies if the Android Gradle plugin is applied to the project and applies the Pitest Gradle plugin if it not yet activated.

## 5.2 Mutant Creation

This section describes the process of creating an APK file for a mutated version of the application. As described in chapter 4 Pimutdroid uses Pitest to create mutated versions of the source code. In order to have access to the byte code an export plugin is used. The class files are exported under using a folder structure which follows the Java package of the classes, as can be seen in figure 5.1. The export is triggered when executing the task “mutateClasses” from section 5.1 as this task depends on the Pitest Gradle task “pitest<buildVariant>”, which runs mutation analysis for local unit tests under “src/test/java”. The build variant (see section 3.2.2.3) is determined from the configured product flavor and build type in Pimutdroid. After Pitest has finished, the markerfiles are created for each mutant class file. To determine whether a mutant was killed by a local unit test, Pimutdroid looks into the XML result “mutations.xml” file from Pitest and verifies the mutation status. When all markerfiles are created the APK files can be

## 5 Implementation Details

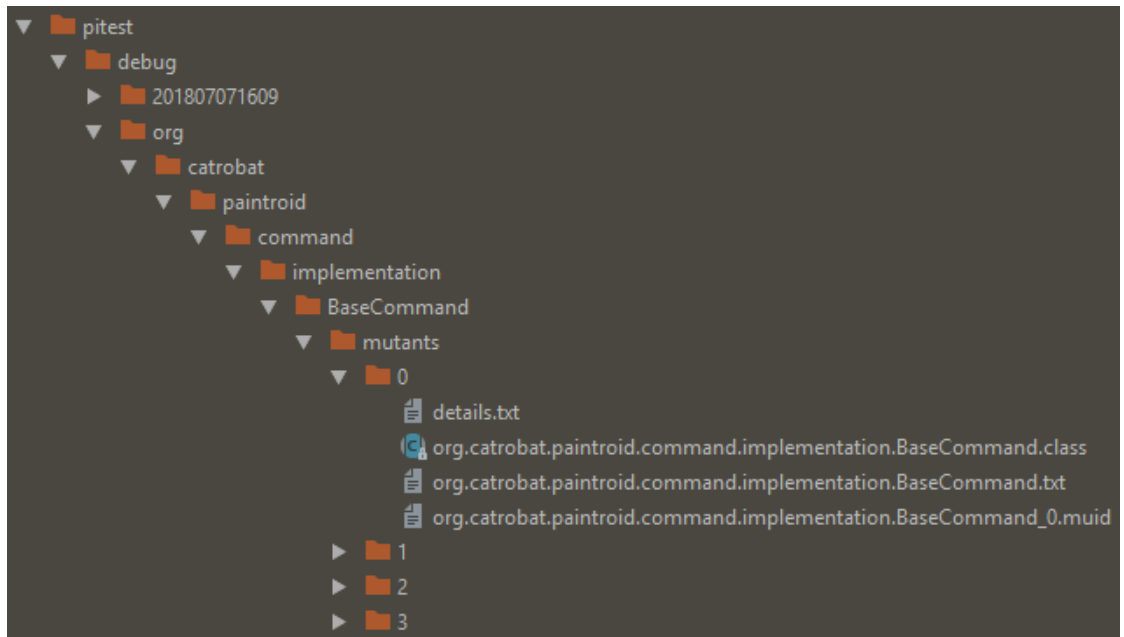


Figure 5.1: Shows the folder structure under which Pitest exports the mutated byte code and metadata files.

assembled.

The task to build the files is named “buildMutantApks”, it determines the Gradle wrapper file from the root project directory and starts a new process for each mutant and runs the task “buildOnlyMutantApk”, using the MUID of the mutant and the Gradle wrapper file (see section 3.1). Additionally for each mutant created the task creates a log file containing the output of the Gradle task. The task “buildOnlyMutantApk” depends on the Android Gradle plugin task to assemble the application APK file “assemble<buildVariant>”. Since the build task is scheduled, Gradle executes the assemble task in addition. The “buildOnlyMutantApk” task is not an ad-hoc task and has the task class “BuildMutantApkTask”, which is used to determine whether a mutant APK file should be created or not. If no task of this specific class is scheduled, the plugin disables the task which is responsible to replace the correct byte code file with the mutated version, as can be seen in listing 5.3. The method “configureTasks” from listing 5.3 is executed when the TEG from section 3.1 is ready. The plugin adds a callback

## 5.2 Mutant Creation

---

```
protected void configureTasks(TaskGraphAdaptor graph) {
    if(graph.hasNotTask(BuildMutantApkTask.class)) {
        LOGGER.debug("Disable replace class with mutant
            class task (no mutant build task found)");
        taskFactory.named(TASK_MUTATE_AFTER_COMPILE_NAME).setEnabled(false);
    }
}
```

---

Listing 5.3: Shows the method “configureTasks” which is executed when the TEG from section 3.1 is ready. It checks if the task has a task of type “BuildMutantApkTask” to handle the build of a mutated APK file. If no mutant APK is build, the task responsible for injecting the mutated byte code file is disabled.

---

```
public void whenTaskGraphReady(final
    Action<TaskGraphAdaptor> readyAction) {
    getTaskGraph().whenReady({ TaskExecutionGraph graph ->
        readyAction.execute(TaskGraphAdaptor.forGraph(graph));
    });
}
```

---

Listing 5.4: Shows the method “whenTaskGraphReady” which executes a passed callback “readyAction” when the TEG from section 3.1 is ready.

to the TEG as can be seen in listing 5.4.

The Android Gradle plugin compiles the Java source files using the task “compile<buildVariant>Sources” as described in section 3.3. Pimutdroid plugs a task into the life cycle of the Android Gradle plugin which executes after the compile sources task and injects the mutated class file. “inject-MutantAfterCompileByMarkerFile” is the name of this task and it takes a MUID as a parameter to determine the mutant it should inject. Afterwards the byte code is transformed to DEX byte code from the Android Gradle plugin and the DEX code as well as the compiled resources are packaged to an APK file. Finally the APK file and the markerfile are copied to the output folder of the mutant and the original byte code files are restored. The folder structure of the mutant as shown in figure 5.2 mirrors the folder structure containing the mutated class files, displayed in figure 5.1.

The approach to replace the files within the task execution speeds up the

## 5 Implementation Details

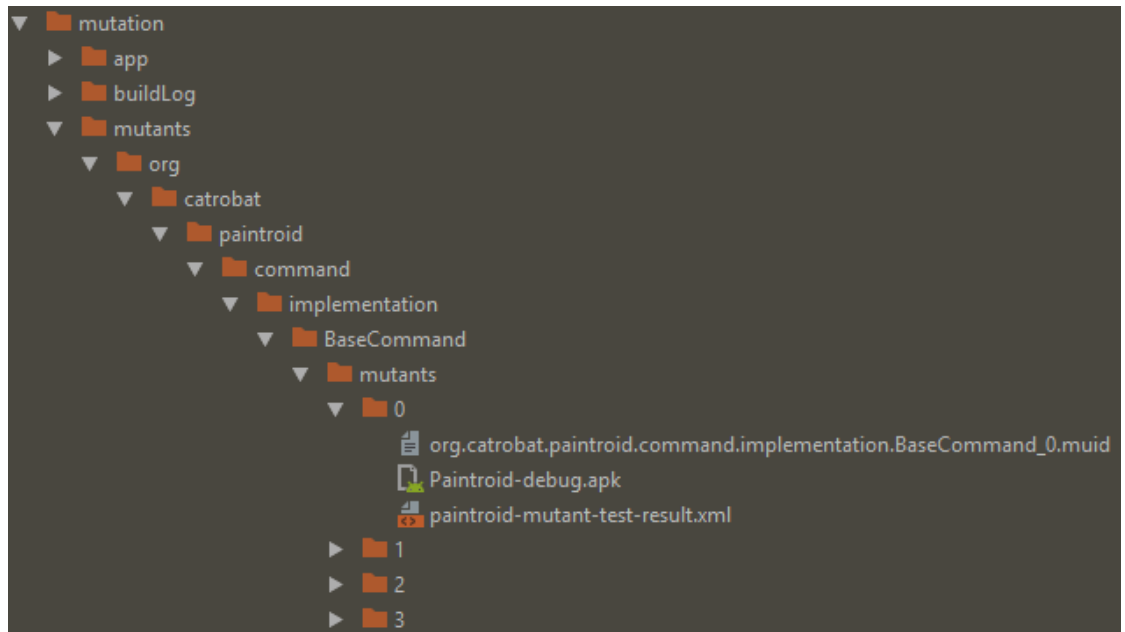


Figure 5.2: Shows the folder structure under which the mutation files per mutant are stored. The plugin mirrors the folder structure containing the mutated class files displayed in figure 5.1

process of mutant creation, as Gradle checks for a task whether the defined input and output files have changed or not. If they changed the task is run, otherwise the task is “up to date” and is skipped (Gradle, 2018b). During the assemble task, only the class files always have to be transformed to DEX files. As after the first mutant APK file created, the other files do not change and the Android tasks are skipped.

### 5.3 Mutant Test Execution

Section 5.3 describes how mutants are tested on an emulator or a real device. Pimutdroid uses the ADB commands described under section 3.4 to install APK files, instrument test cases, fetch result files and determines the devices to use. Additionally the plugin adds a dependency to the

---

```

int numMutants = mutantApks.size();
int numDevices = deviceList.size();
List<String> fullMutantApkFilepathList =
    mutantApks.collect({ File file ->
        file.getPath().toString() }).toList();

def mutantPartition =
    getMutantPathsPerDevice(fullMutantApkFilepathList,
        numDevices);

```

---

Listing 5.5: Shows how the targeted APK files are partitioned for a the number of devices, on test execution.

test APK build which is an implementation of an Android instrumentation run listener by Schröpf (2018). This run listener creates a JUnit compatible XML report file which is used for the expected and mutant result files. The listener is added to the ADB instrumentation command by supplying the listener class using the argument “-e listener <full class name>”. The XML result file is stored on the device under “/storage/emulated/o/Android/data/<application package>/files” and has the name “report-o.xml” by default. In case of Paintroid the file is located under “/storage/emulated/o/Android/data/org.catrobat.paintroid/files”.

Using the ADB query devices command from section 3.4.1 and the “devices” plugin configuration as explained in section 4.3.2, the devices for test execution are determined. The targeted mutant APK files are then evenly partitioned among the devices using the Groovy method “collate”<sup>1</sup>, as can be seen in listing 5.5 and listing 5.6. A list of mutant APK files which should be tested is passed to each device.

The workflow to install the mutants and run the tests for each device consists of the following steps:

- Install and replace test APK file using the ADB command to install an APK file with the “-t” option, see section 3.4.2.2.

---

<sup>1</sup>[http://docs.groovy-lang.org/2.4.3/html/groovy-jdk/java/lang/Iterable.html#collate\(int\)](http://docs.groovy-lang.org/2.4.3/html/groovy-jdk/java/lang/Iterable.html#collate(int))

## 5 Implementation Details

---

```
private List<List<String>>
getMutantPathsPerDevice(List<String>
fullMutantApkFilepathList, int numDevices) {
    int numMutants = fullMutantApkFilepathList.size();
    int partitionSize = (int)(numMutants / numDevices);
    int remainderSize = (numMutants % numDevices);

    List<List<String>> mutantPartition =
        fullMutantApkFilepathList.collate(partitionSize);
    List<String> remainderList = (remainderSize > 0) ?
        mutantPartition.pop() : new ArrayList<String>();

    remainderList.eachWithIndex { String path, int index ->
        mutantPartition.get(index).add(path);
    }

    return mutantPartition;
}
```

---

Listing 5.6: Shows the method “getMutantPathsPerDevice” which partitions a given list of APK file paths for a given number of devices using the Groovy function “collate”.

- Remove any existing XML result file created by the run listener from the device.
- For each mutant:
  1. Install and replace the current installation of the mutated application APK file. For this ADB call a timeout of four minutes exists, if the time runs out the test for this APK file is aborted, the mutant outcome will be “NO\_RESULT” and counts as not killed.
  2. Run tests using instrument command from section 3.4.2.6. The process waits until test execution finishes and after this step the XML result file is created. As described in 4.3.2 the default test timeout is set to 10 seconds.
  3. Copy mutant result file from the device to the mutant folder, see figure 5.2.

## 5.3 Mutant Test Execution

```
...
Partition mutants 54 on 2 devices.
Submit worker for device 'emulator-5554'... work on '27'
  mutants
Submit worker for device 'emulator-5556'... work on '27'
  mutants

On device 'emulator-5554': (1/27)|(took: 00m:03s)|(tests
took: 00m:01s) - finished testing
  'E:\mutants\org\catrobat\paintroid\command\UndoRedoManager
  \mutants\0\Paintroid-debug.apk'
(inst: true, test: true, fetch: true, remove: true, clear
test: true, clear app: true)
On device 'emulator-5556': (1/27)|(took: 00m:43s)|(tests
took: 00m:41s) - finished testing
  'E:\mutants\org\catrobat\paintroid\command\UndoRedoManager
  \mutants\33\Paintroid-debug.apk'
...
```

Listing 5.7: Shows the log output by the “testMutants” task, which displays how long test execution for a specific APK file took and if the test commands were successful.

4. Remove test result file on device.
  5. Clear package data of the test package.
  6. Clear package data of the application.
- Finally, when all mutants are tested, uninstall application APK file.

The task to start the test execution is named “testMutants” and waits until all tests finished on the configured devices. For each mutant tested a log output is printed, showing how long the complete test took and how long the tests ran as can be seen in listing 5.7. Using this log it is possible to re-run failed test execution for certain mutants.





## 6 Conclusion

The thesis shows that the developed plugin Pimutdroid is easy to integrate into the current Android build system and that it enables to run mutation analysis on local unit tests as well as instrumented test cases. It offers several options to speed up the mutation analysis process but it depends on the actual test set. The mutation analysis tool was used to evaluate the test set of the Paintroid project. The performed test analysis shows the lack of local unit tests, which would be also important for faster feedback on changes of the program. As described in section 2.4, Espresso tests are heavily used in the test set. They tend to be slow and unstable, depending on execution order as well as on the device the tests are executed on. Based on this information, the plugin option “ignoreKilledByUnitTest” was implemented and the tool offers the ability to skip mutants which are already killed by local unit tests. This provides the possibility to implement local unit tests in order to reduce the number of mutants which must be assembled into an APK file and be tested on an emulator or a real device. Using Pimutdroid to run mutation analysis while creating these local unit tests helps in writing effective test cases. It should be used in the Paintroid project to build up an effective base of local unit tests which kill most of the mutants created by Pitest and only tests which increase the mutation score should be added to the test set. Subsequent work needs to be done to write more real instrumented unit test cases and remove the Espresso APIs from the existing tests. Likewise this should be supported by mutation testing to guarantee robust test cases. Mutants which then can not be killed by test set should be run against the UI tests. For each mutant, Pimutdroid stores the test result which can be used to identify test cases which are able to detect mutants and which are not. This could be used to reduce the number of Espresso tests by removing test cases which do not kill any mutant and hence do not decrease the mutation score. Earlier runs of the mutation analysis on the mutant set of Paintroid ended in a hung up test case, as problems with infinite loops

## 6 Conclusion

resulting from mutations were previously not detected. The Android Junit runner used in the execution process offers a configuration option to set a timeout for tests, Pimutdroid uses this option and sets it by default to 10 seconds, though it can be configured in the projects build script. Another problem was the fact that sometimes the ADB install command did not finish and stalled the test process on a device. For this situation another timeout was introduced by Pimutdroid which aborts the install command, by default the timeout triggers after four minutes, to guarantee that test execution finishes.

During the use of Pimutdroid the problem emerged that Paintroid allocates resources such as bitmaps, as it is a drawing application and has several background threads and asynchronous tasks, but does not free all resources correctly. This problem is present in the application as well as test cases. Espresso launches an activity for every test which led to the problem that based on the resource allocation after several test runs, the used emulator got significantly slower and that the test execution time increased with every mutant tested. After some hours system applications on the emulator will stop working and the emulator hangs up. Using the default mutation operators of Pitest, Pimutdroid created 3995 mutants of which 35 were killed by local unit tests. Running the whole espresso test set takes about 14 minutes which would result in an execution time on a single emulator of about 38 days for the complete test set. Execution of the instrumented unit tests would take, if they are able to run in about 40 seconds on average, about 1.5 days. However, 10 test classes of the 17 instrumented unit test classes use Espresso APIs which again resulted in an increasing test execution time. The seven “real” instrumented unit tests had the problem that they allocated Android bitmaps for each test case and likewise were not released properly, which again resulted in an emulator that stalled. For these test cases the resource allocation was fixed and then they ran on average in 1 second on a device, with installation of the mutant APK file and gathering of the result, on average in about seven seconds. To reduce the amount of mutants, two of the seven default mutation operators were not used. The five used operators were the increments, return values, negate conditional, invert negatives and conditional boundary mutators. Resulting in 1460 mutants for 76 classes of which 18 were killed by local unit tests. The math mutation operator created 961 mutants of which one was killed and the void method calls

mutator gained a mutation score of only one percent, as of the 1574 mutants created only 16 were killed by the local unit tests.

Using the “ignoreKilledByUnitTest” option, 1442 mutated APK files of the 1460 mutants were assembled with Pimutdroid. This took about 2 hours and 9 seconds to finish, as the plugin utilizes the incremental build feature of Gradle, it takes about only five seconds to assemble the APK file. Subsequently the seven instrumented unit tests were assembled into an APK file and were executed in parallel on 2 emulators, using the parallel test execution configuration. It took 5 hours and 31 minutes to finish test execution. 51 mutants were killed by the test set, from which 18 were already killed by local unit tests. Resulting in a mutation score of 3.5%. Tests were written for the “org.catrobat.paintroid.command.implementation” package and 30 mutants of this package were killed. Two mutants of the class “FileIO” of package “org.catrobat.paintroid” were killed as the mutation resulted in an infinite loop of the program execution, which the used test timeout detected. One more mutant of package “org.catrobat.paintroid.tools” for class “Layer” was killed by the instrumented unit tests.

Finally, additional work on the plugin should be done, however they were beyond the scope of this thesis, hence they were not covered. Pimutdroid should implement some of the proposed Android specific mutation operators by Lin Deng et al. (2017b) and Linares-Vásquez et al. (2017), to better simulate faults which are specific for Android applications. Pitest supports the addition of custom mutation operators to the set of available operators, however the mutation has to be applied to the byte code. Alternatively, the mutation framework by Linares-Vásquez et al. (2017) could be used to generate Android specific mutants and run with Pimutdroid, as only the markerfile must be created for the APK file in order to be able to run it. Additional work needs to be done for the test execution to better handle memory and resource related issues on test execution. For example, a restart of the emulator after several mutants have been tested, to free any allocated resources. Pimutdroid should offer a mutation report in HTML format as well to give a better overview of the mutation outcome, it could be combined with the report created by Pitest. Additionally, Android supports to gather information on line coverage for instrumented test cases, this data could be linked to the report created by Pimutdroid to add additional information to the report. So that the outcome of a mutant does not only

## 6 Conclusion

indicate whether it was killed or not, but also includes if the mutant was reached and survived or the mutant was not even covered by the test set. Furthermore Pimutdroid could be used to perform high-order mutation and inject not only one mutated class file but multiple at once into an APK file.

### Final Words From The Author

There are many ways to improve the current plugin, unfortunately the test set of Paintroid consisted of so many Espresso test cases. But I hope that Pimutdroid will aid in the process of improving or implementing the local and instrumented unit tests of the project. Although it does not support Android specific mutation operators, it will be a good start to kill mutants created from the traditional Java operators. In retrospect it would have been great if support for XML mutation operators got included in the current plugin, to test the effect on the Android build process and which additional Gradle tasks would have been needed. However mutation testing is a great method to improve ones test cases and additionally improve yourself on how to write such test cases.

# Appendix



# Bibliography

- Ammann, Paul and Jeff Offutt (2008). *Introduction to Software Testing*. 1st ed. New York, NY, USA: Cambridge University Press. ISBN: 0521880386, 9780521880381 (cit. on pp. 3, 12).
- Andrews, J. H., L. C. Briand, and Y. Labiche (May 2005). "Is mutation an appropriate tool for testing experiments? [software testing]." In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. Pp. 402–411. DOI: [10.1109/ICSE.2005.1553583](https://doi.org/10.1109/ICSE.2005.1553583) (cit. on p. 6).
- Andrews, J. H., L. C. Briand, Y. Labiche, and A. S. Namin (Aug. 2006). "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria." In: *IEEE Transactions on Software Engineering* 32.8, pp. 608–624. ISSN: 0098-5589. DOI: [10.1109/TSE.2006.83](https://doi.org/10.1109/TSE.2006.83) (cit. on p. 6).
- Catrobat (2018). *Free educational apps for children and teenagers*. URL: <https://www.catrobat.org/> (visited on 07/01/2018) (cit. on p. 21).
- Cohen, Mike (2009). *The Forgotten Layer of the Test Automation Pyramid*. URL: <https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid> (visited on 05/27/2018) (cit. on p. 21).
- Coles, Henry (2018a). *Github: Pitest MutationDetails class*. URL: <https://github.com/hcoles/pitest/blob/pitest-parent-1.2.2/pitest/src/main/java/org/pitest/mutationtest/engine/MutationDetails.java> (visited on 07/05/2018) (cit. on p. 50).
- Coles, Henry (2018b). *Github: Pitest MutationIdentifier class*. URL: <https://github.com/hcoles/pitest/blob/pitest-parent-1.2.2/pitest/src/main/java/org/pitest/mutationtest/engine/MutationIdentifier.java> (visited on 07/05/2018) (cit. on p. 50).
- Coles, Henry (2018c). *PIT*. URL: <http://pitest.org/> (visited on 05/15/2018) (cit. on pp. 17, 43, 52).
- Coles, Henry (2018d). *PIT Export Plugin*. URL: <https://github.com/pitest/export-plugin> (visited on 05/15/2018) (cit. on pp. 44, 47, 58).

## Bibliography

- Coles, Henry (2018e). *State of the art mutation testing system for the JVM*. URL: <https://github.com/hcoles/pitest> (visited on 05/25/2018) (cit. on p. 17).
- Coles, Henry and Stefan Penndorf (2018a). *PIT Conditionals Boundary Mutator*. URL: [http://pitest.org/quickstart/mutators/#CONDITIONALS\\_BOUNDARY](http://pitest.org/quickstart/mutators/#CONDITIONALS_BOUNDARY) (visited on 05/16/2018) (cit. on p. 52).
- Coles, Henry and Stefan Penndorf (2018b). *PIT Constructor Call Mutator*. URL: [http://pitest.org/quickstart/mutators/#CONSTRUCTOR\\_CALLS](http://pitest.org/quickstart/mutators/#CONSTRUCTOR_CALLS) (visited on 05/16/2018) (cit. on p. 56).
- Coles, Henry and Stefan Penndorf (2018c). *PIT Increments Mutator*. URL: <http://pitest.org/quickstart/mutators/#INCREMENTS> (visited on 05/16/2018) (cit. on p. 53).
- Coles, Henry and Stefan Penndorf (2018d). *PIT Inline Constant Mutator*. URL: [http://pitest.org/quickstart/mutators/#INLINE\\_CONSTS](http://pitest.org/quickstart/mutators/#INLINE_CONSTS) (visited on 05/16/2018) (cit. on p. 57).
- Coles, Henry and Stefan Penndorf (2018e). *PIT Invert Negatives Mutator*. URL: [http://pitest.org/quickstart/mutators/#INVERT\\_NEGS](http://pitest.org/quickstart/mutators/#INVERT_NEGS) (visited on 05/16/2018) (cit. on p. 53).
- Coles, Henry and Stefan Penndorf (2018f). *PIT Math Mutator*. URL: <http://pitest.org/quickstart/mutators/#MATH> (visited on 05/16/2018) (cit. on pp. 53, 54).
- Coles, Henry and Stefan Penndorf (2018g). *PIT Mutators*. URL: <http://pitest.org/quickstart/mutators/> (visited on 05/16/2018) (cit. on pp. 51, 52).
- Coles, Henry and Stefan Penndorf (2018h). *PIT Negate Conditionals Mutator*. URL: [http://pitest.org/quickstart/mutators/#NEGATE\\_CONDITIONALS](http://pitest.org/quickstart/mutators/#NEGATE_CONDITIONALS) (visited on 05/16/2018) (cit. on pp. 54, 55).
- Coles, Henry and Stefan Penndorf (2018i). *PIT Non Void Method Call Mutator*. URL: [http://pitest.org/quickstart/mutators/#NON\\_VOID\\_METHOD\\_CALLS](http://pitest.org/quickstart/mutators/#NON_VOID_METHOD_CALLS) (visited on 05/16/2018) (cit. on p. 56).
- Coles, Henry and Stefan Penndorf (2018j). *PIT Remove Conditionals Mutator*. URL: [http://pitest.org/quickstart/mutators/#REMOVE\\_CONDITIONALS](http://pitest.org/quickstart/mutators/#REMOVE_CONDITIONALS) (visited on 05/16/2018) (cit. on p. 57).
- Coles, Henry and Stefan Penndorf (2018k). *PIT Return Values Mutator*. URL: [http://pitest.org/quickstart/mutators/#RETURN\\_VALS](http://pitest.org/quickstart/mutators/#RETURN_VALS) (visited on 05/16/2018) (cit. on pp. 54, 55).



- Coles, Henry and Stefan Penndorf (2018l). *PIT Void Method Call Mutator*. URL: [http://pitest.org/quickstart/mutators/#VOID\\_METHOD\\_CALLS](http://pitest.org/quickstart/mutators/#VOID_METHOD_CALLS) (visited on 05/16/2018) (cit. on p. 55).
- Daran, Murial and Pascale Thévenod-Fosse (1996a). "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations." In: *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '96. San Diego, California, USA: ACM, pp. 158–171. ISBN: 0-89791-787-1. DOI: [10.1145/229000.226313](https://doi.org/10.1145/229000.226313). URL: <http://doi.acm.org/10.1145/229000.226313> (cit. on p. 6).
- Daran, Murial and Pascale Thévenod-Fosse (May 1996b). "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations." In: *SIGSOFT Softw. Eng. Notes* 21.3, pp. 158–171. ISSN: 0163-5948. DOI: [10.1145/226295.226313](https://doi.org/10.1145/226295.226313). URL: <http://doi.acm.org/10.1145/226295.226313> (cit. on p. 6).
- Delahaye, M. and L. du Bousquet (July 2013). "A Comparison of Mutation Analysis Tools for Java." In: *2013 13th International Conference on Quality Software*, pp. 187–195. DOI: [10.1109/QSIC.2013.47](https://doi.org/10.1109/QSIC.2013.47) (cit. on pp. 16, 22).
- DeMillo, R. A., R. J. Lipton, and F. G. Sayward (Apr. 1978). "Hints on Test Data Selection: Help for the Practicing Programmer." In: *Computer* 11.4, pp. 34–41. ISSN: 0018-9162. DOI: [10.1109/C-M.1978.218136](https://doi.org/10.1109/C-M.1978.218136) (cit. on p. 5).
- Deng, L., N. Mirzaei, et al. (Apr. 2015). "Towards mutation analysis of Android apps." In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–10. DOI: [10.1109/ICSTW.2015.7107450](https://doi.org/10.1109/ICSTW.2015.7107450) (cit. on pp. 12, 14, 22, 23, 25, 26, 46).
- Deng, L., J. Offutt, and D. Samudio (July 2017). "Is Mutation Analysis Effective at Testing Android Apps?" In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 86–93. DOI: [10.1109/QRS.2017.19](https://doi.org/10.1109/QRS.2017.19) (cit. on p. 25).
- Deng, Lin et al. (2017a). *Mutation Operators for Testing Android Apps*. URL: <https://pdfs.semanticscholar.org/5afa/a2dfade323a433056e294dcc9bf8b0173cf4.pdf> (visited on 05/24/2018) (cit. on p. 25).
- Deng, Lin et al. (Jan. 2017b). "Mutation Operators for Testing Android Apps." In: *Inf. Softw. Technol.* 81.C, pp. 154–168. ISSN: 0950-5849. DOI: [10.1016/j.infsof.2016.04.012](https://doi.org/10.1016/j.infsof.2016.04.012). URL: <https://doi.org/10.1016/j.infsof.2016.04.012> (cit. on pp. 25, 87).
- Fowler, Martin (2012). *TestPyramid*. URL: <https://martinfowler.com/bliki/TestPyramid.html> (visited on 05/27/2018) (cit. on p. 21).

## Bibliography

- Geist, R., A. J. Offutt, and F. C. Harris (May 1992). "Estimation and enhancement of real-time software reliability through mutation analysis." In: *IEEE Transactions on Computers* 41.5, pp. 550–558. ISSN: 0018-9340. DOI: 10.1109/12.142681 (cit. on p. 12).
- Google (2010). *AndroidJUnitRunner documentation*. URL: <https://developer.android.com/reference/android/support/test/runner/AndroidJUnitRunner> (visited on 05/27/2018) (cit. on pp. 19, 63).
- Google (2018a). *Android Activity*. URL: <https://developer.android.com/reference/android/app/Activity> (visited on 06/01/2018) (cit. on p. 55).
- Google (2018b). *Android API reference*. URL: <https://developer.android.com/reference/> (visited on 06/01/2018) (cit. on p. 18).
- Google (2018c). *Android Debug Bridge (adb)*. URL: <https://developer.android.com/studio/command-line/adb> (visited on 05/17/2018) (cit. on p. 37).
- Google (2018d). *Android Plugin DSL Reference*. URL: <http://google.github.io/android-gradle-dsl/3.1/> (visited on 06/01/2018) (cit. on pp. 31, 47, 48).
- Google (2018e). *Testing Documentation*. URL: <https://developer.android.com/training/testing/> (visited on 06/01/2018) (cit. on p. 17).
- Google (2018f). *Testing Documentation*. URL: <https://developer.android.com/training/testing/fundamentals#on-device-unit-tests> (visited on 06/01/2018) (cit. on p. 20).
- Google (2018g). *User Guide: Configure build variants*. URL: <https://developer.android.com/studio/build/build-variants> (visited on 06/01/2018) (cit. on p. 32).
- Google (2018h). *User Guide: Configure your build*. URL: <https://developer.android.com/studio/build/> (visited on 06/01/2018) (cit. on pp. 30, 36).
- Google (2018i). *User Guide: Set the application ID*. URL: <https://developer.android.com/studio/build/application-id> (visited on 06/01/2018) (cit. on p. 36).
- Gradle (2018a). *Gradle Build System*. URL: <https://docs.gradle.org/> (visited on 05/17/2018) (cit. on p. 27).
- Gradle (2018b). *Gradle Docs: Up-to-date checks (AKA Incremental Build)*. URL: [https://docs.gradle.org/4.8.1/userguide/more\\_about\\_tasks.html#sec:up\\_to\\_date\\_checks](https://docs.gradle.org/4.8.1/userguide/more_about_tasks.html#sec:up_to_date_checks) (visited on 05/27/2018) (cit. on p. 80).
- Gradle (2018c). *Gradle User Manual*. URL: <https://docs.gradle.org/4.8.1/userguide/userguide.html> (visited on 05/27/2018) (cit. on p. 27).

- Gradle Wrapper (2018). *Gradle Docs: The Gradle Wrapper*. URL: [https://docs.gradle.org/4.8.1/userguide/gradle\\_wrapper.html](https://docs.gradle.org/4.8.1/userguide/gradle_wrapper.html) (visited on 05/27/2018) (cit. on p. 29).
- Groovy (2018). *Groovy: A multi-faceted language for the Java platform*. URL: <http://www.groovy-lang.org/> (visited on 07/02/2018) (cit. on pp. 27, 44).
- Jackson (2018). *Extension for Jackson JSON processor that adds support for serializing POJOs as XML (and deserializing from XML) as an alternative to JSON*. URL: <https://github.com/FasterXML/jackson-dataformat-xml> (visited on 07/07/2018) (cit. on p. 58).
- Java (2018a). *Java: Equality, Relational, and Conditional Operators*. URL: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html> (visited on 06/23/2018) (cit. on p. 7).
- Java (2018b). *Java*. URL: <https://www.java.com/> (visited on 07/02/2018) (cit. on p. 44).
- Jia, Y. and M. Harman (Sept. 2011). “An Analysis and Survey of the Development of Mutation Testing.” In: *IEEE Transactions on Software Engineering* 37:5, pp. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62 (cit. on pp. 7, 15, 52).
- Just, R., G. M. Kapfhammer, and F. Schweiggert (Apr. 2012). “Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?” In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 720–725. DOI: 10.1109/ICST.2012.162 (cit. on pp. 7, 14, 15, 52).
- Just, R., F. Schweiggert, and G. M. Kapfhammer (Nov. 2011). “MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler.” In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 612–615. DOI: 10.1109/ASE.2011.6100138 (cit. on p. 17).
- Just, René (2018). *The Major mutation framework*. URL: <http://mutation-testing.org/> (visited on 05/22/2018) (cit. on p. 17).
- Just, René (2014a). “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. San Jose, CA, USA: ACM, pp. 433–436. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2628053. URL: <http://doi.acm.org/10.1145/2610384.2628053> (cit. on p. 17).

## Bibliography

- Just, René (2014b). *The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java*. URL: [https://people.cs.umass.edu/~rjust/publ/major\\_issta\\_2014.pdf](https://people.cs.umass.edu/~rjust/publ/major_issta_2014.pdf) (visited on 05/22/2018) (cit. on p. 17).
- Just, René et al. (2014). "Are Mutants a Valid Substitute for Real Faults in Software Testing?" In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, pp. 654–665. ISBN: 978-1-4503-3056-5. DOI: [10.1145/2635868.2635929](https://doi.org/10.1145/2635868.2635929). URL: <http://doi.acm.org/10.1145/2635868.2635929> (cit. on p. 6).
- Kaminski, Gary, Paul Ammann, and Jeff Offutt (2011). "Better Predicate Testing." In: *Proceedings of the 6th International Workshop on Automation of Software Test*. AST '11. Waikiki, Honolulu, HI, USA: ACM, pp. 57–63. ISBN: 978-1-4503-0592-1. DOI: [10.1145/1982595.1982608](https://doi.org/10.1145/1982595.1982608). URL: <http://doi.acm.org/10.1145/1982595.1982608> (cit. on pp. 15, 52, 54).
- Kotlin (2018). *Kotlin: Statically typed programming language for modern multiplatform applications*. URL: <https://kotlinlang.org/> (visited on 07/02/2018) (cit. on p. 27).
- Linares-Vásquez, M. et al. (July 2017). "Enabling Mutation Testing for Android Apps." In: *ArXiv e-prints*. arXiv: [1707.09038 \[cs.SE\]](https://arxiv.org/abs/1707.09038) (cit. on pp. 23, 26, 87).
- Ma, Yu-Seung and Jeff Offutt (2006). *Description of muJava's Method-level Mutation Operators*. URL: <https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf> (visited on 06/23/2018) (cit. on p. 7).
- Ma, Yu-Seung, Jeff Offutt, and Yong Rae Kwon (June 2005). "MuJava: An Automated Class Mutation System: Research Articles." In: *Softw. Test. Verif. Reliab.* 15.2, pp. 97–133. ISSN: 0960-0833. DOI: [10.1002/stvr.v15:2](https://doi.org/10.1002/stvr.v15:2). URL: <http://dx.doi.org/10.1002/stvr.v15:2> (cit. on p. 24).
- Ma, Yu-Seung, Jeff Offutt, and Yong-Rae Kwon (2006). "MuJava: A Mutation System for Java." In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, pp. 827–830. ISBN: 1-59593-375-1. DOI: [10.1145/1134285.1134425](https://doi.org/10.1145/1134285.1134425). URL: <http://doi.acm.org/10.1145/1134285.1134425> (cit. on pp. 7, 17).
- Madeyski, L. and N. Radyk (Feb. 2010). "Judy - a mutation testing tool for java." In: *IET Software* 4.1, pp. 32–42. ISSN: 1751-8806. DOI: [10.1049/iet-sen.2008.0038](https://doi.org/10.1049/iet-sen.2008.0038) (cit. on p. 17).
- Mateo, P. R. and M. P. Usaola (Sept. 2012). "Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases." In: *2012 28th IEEE*

- International Conference on Software Maintenance (ICSM)*, pp. 646–649. DOI: [10.1109/ICSM.2012.6405344](https://doi.org/10.1109/ICSM.2012.6405344) (cit. on p. 17).
- Mateo, P. R., M. P. Usaola, and J. Offutt (Apr. 2010). “Mutation at System and Functional Levels.” In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 110–119. DOI: [10.1109/ICSTW.2010.18](https://doi.org/10.1109/ICSTW.2010.18) (cit. on p. 17).
- Moore, Ivan (2001). “Jester - a JUnit test tester.” 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP). Villasimius, Italy. URL: <http://ciclamino.dibe.unige.it/xp2001/conference/papers/Chapter20-Moore.pdf> (visited on 05/24/2018) (cit. on p. 17).
- Moore, Ivan (2013). *Jester Sourceforge Project Page*. URL: <https://sourceforge.net/projects/jester/> (visited on 05/24/2018) (cit. on p. 17).
- Moran, K. et al. (Feb. 2018). “MDroid+: A Mutation Testing Framework for Android.” In: *ArXiv e-prints*. arXiv: [1802.04749](https://arxiv.org/abs/1802.04749) [cs.SE] (cit. on p. 26).
- Morell, L. J. (Aug. 1990). “A theory of fault-based testing.” In: *IEEE Transactions on Software Engineering* 16.8, pp. 844–857. ISSN: 0098-5589. DOI: [10.1109/32.57623](https://doi.org/10.1109/32.57623) (cit. on pp. 4, 5).
- Myers, Glenford J. (1979). *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0471043281 (cit. on p. 4).
- Offutt, A. J. and Jie Pan (June 1996). “Detecting equivalent mutants and the feasible path problem.” In: *Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*, pp. 224–236. DOI: [10.1109/CMPASS.1996.507890](https://doi.org/10.1109/CMPASS.1996.507890) (cit. on pp. 12, 13).
- Offutt, A. Jefferson (Jan. 1992a). *Investigations of the Software Testing Coupling Effect*. URL: <https://cs.gmu.edu/~offutt/rsrch/papers/coupl.pdf> (visited on 05/24/2018) (cit. on pp. 5, 6).
- Offutt, A. Jefferson (Jan. 1992b). “Investigations of the Software Testing Coupling Effect.” In: *ACM Trans. Softw. Eng. Methodol.* 1.1, pp. 5–20. ISSN: 1049-331X. DOI: [10.1145/125489.125473](https://doi.org/10.1145/125489.125473). URL: <http://doi.acm.org/10.1145/125489.125473> (cit. on pp. 5, 6).
- Offutt, A. Jefferson and Ronald H. Untch (2001a). *Mutation 2000: Uniting the Orthogonal*. URL: <https://cs.gmu.edu/~offutt/rsrch/papers/mut00.pdf> (visited on 05/24/2018) (cit. on pp. 3, 7).
- Offutt, A. Jefferson and Ronald H. Untch (2001b). “Mutation Testing for the New Century.” In: ed. by W. Eric Wong. Norwell, MA, USA: Kluwer



## Bibliography

- Academic Publishers. Chap. Mutation 2000: Uniting the Orthogonal, pp. 34–44. ISBN: 0-7923-7323-5. URL: <http://dl.acm.org/citation.cfm?id=571305.571314> (cit. on pp. 3, 7, 9, 12, 15).
- Offutt, Jeff (2015). *Mutation system for Java programs, including OO mutation operators*. URL: <https://github.com/jeffoffutt/muJava/> (visited on 05/22/2018) (cit. on p. 17).
- Pocket Paint (2018a). *GitHub Pocket Paint project: The standard image manipulation app for Catroid*. URL: <https://github.com/Catrobat/Paintroid> (visited on 07/01/2018) (cit. on p. 21).
- Pocket Paint (2018b). *Pocket Paint: draw and edit!* URL: <https://play.google.com/store/apps/details?id=org.catrobat.paintroid> (visited on 07/01/2018) (cit. on p. 21).
- Robolectric (2010). *Robolectric: test-drive your Android code*. URL: <http://robolectric.org/> (visited on 05/27/2018) (cit. on p. 19).
- Robotium (2010). *Android UI Testing*. URL: <http://www.robotium.org> (visited on 05/27/2018) (cit. on pp. 18, 21, 24).
- Schröpf, Tobias (2018). *An AndroidJUnitRunner RunListener implementation which will create JUnit compatible XML report files containing the results of Android Instrumentation tests*. URL: <https://github.com/schroepf/TestLab/tree/master/android> (visited on 05/14/2018) (cit. on p. 81).
- Schuler, David (2009). *Javalanche: Efficient Mutation Testing for Java*. URL: <https://github.com/david-schuler/javalanche> (visited on 05/24/2018) (cit. on p. 17).
- Schuler, David and Andreas Zeller (2009a). “Javalanche: Efficient Mutation Testing for Java.” In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. Amsterdam, The Netherlands: ACM, pp. 297–298. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595750. URL: <http://doi.acm.org/10.1145/1595696.1595750> (cit. on p. 17).
- Schuler, David and Andreas Zeller (2009b). *Javalanche: Efficient Mutation Testing for Java*. URL: <http://javalanche.org> (visited on 05/24/2018) (cit. on p. 17).
- Tai, Wah K. S. How (1995). “Fault coupling in finite bijective functions.” In: *Software Testing, Verification and Reliability* 5.1, pp. 3–47. DOI: 10.1002/stvr.4370050103. eprint: <https://onlinelibrary.wiley.com/doi/pdf/>

- 10.1002/stvr.4370050103. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.4370050103> (cit. on p. 6).
- Tai, Wah K. S. How (2000). "A theoretical study of fault coupling." In: *Software Testing, Verification and Reliability* 10.1, pp. 3–45. DOI: 10.1002/(SICI)1099-1689(200003)10:1<3::AID-STVR196>3.0.CO;2-P. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291099-1689%28200003%2910%3A1%3C3%3A%3AAID-STVR196%3E3.0.CO%3B2-P>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-1689%28200003%2910%3A1%3C3%3A%3AAID-STVR196%3E3.0.CO%3B2-P> (cit. on p. 6).
- Utting, Mark and Len Trigg (2007). *Jumble Sourceforge Project Page*. URL: <https://sourceforge.net/projects/jumble/> (visited on 05/22/2018) (cit. on p. 17).
- Wrotniak, Karol (2018). *Gradle plugin for PIT Mutation Testing in Android projects*. URL: <https://github.com/koral--/gradle-pitest-plugin> (visited on 05/15/2018) (cit. on pp. 23, 43, 58).
- Wu, D. et al. (July 1988). "A practical method for software quality control via program mutation." In: [1988] *Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pp. 159–170. DOI: 10.1109/WST.1988.5371 (cit. on p. 7).
- Yuan, Wei (2016a). *Github project of MuDroid: Mutation Testing tool for Android Integration Testing*. URL: <https://github.com/Yuan-W/muDroid> (visited on 05/17/2018) (cit. on p. 22).
- Yuan, Wei (2016b). *MuDroid: Mutation Testing for Android Apps*. URL: [http://www0.cs.ucl.ac.uk/staff/Yue.Jia/resources/studentprojects/Yuan\\_Wei\\_MuDroid\\_Mutation\\_Testing\\_for\\_Android\\_Apps.pdf](http://www0.cs.ucl.ac.uk/staff/Yue.Jia/resources/studentprojects/Yuan_Wei_MuDroid_Mutation_Testing_for_Android_Apps.pdf) (visited on 05/17/2018) (cit. on p. 23).
- Zajączkowski, Marcin (2018). *Gradle plugin for PIT Mutation Testing*. URL: <https://gradle-pitest-plugin.solidsoft.info/> (visited on 05/15/2018) (cit. on pp. 17, 23, 44).