4-2018

# What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications

Anthony Shehan Ayam Peruma
axp6201@rit.edu

# What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications

by

**Anthony Shehan Ayam Peruma**

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Mohamed Wiem Mkaouer

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

April 2018

The thesis “What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications” by Anthony Shehan Ayam Peruma has been examined and approved by the following Examination Committee:

Dr. Mohamed Wiem Mkaouer
Assistant Professor
Thesis Committee Chair

Dr. Christian D. Newman
Assistant Professor

Dr. Mehdi Mirakhorli
Assistant Professor

Dr. J. Scott Hawker
Associate Professor
Graduate Program Director

*To my family and friends, for all of their endless love, support, and encouragement*

*To Sunshine & Kimi - my number 1 fans!*

$\infty$

# Acknowledgments

Accomplishing a work of this magnitude would not be possible single-handedly. I am eternally grateful to the many individuals who supported, advised and encouraged me.

My sincerest thanks to my advisor, Dr. Mohamed Wiem Mkaouer, for his constant dedication and guidance throughout my research. His vast knowledge and experience provided me with a solid foundation to conduct my research and instilled in me an appreciation of the benefits of research in software quality. I am privileged and ever grateful, to the faculty members of the Department of Software Engineering for exposing me to many different areas of software engineering and providing insight on the different strategies and techniques for conducting, evaluating and reporting on research activities.

I would be failing in my duties if I forget to acknowledge my friends and colleagues who provided me with a constant source of encouragement throughout my duration of my Master's program. Last, but not least, my family - thank you for all your encouragement and support that empowered me to pursue my Master of Science in Software Engineering.

# Abstract

The widespread adoption of mobile devices, coupled with the ease of developing mobile-based applications (apps) has created a lucrative and competitive environment for app developers. Solely focusing on app functionality and time-to-market is not enough for developers to ensure the success of their app. Quality attributes exhibited by the app must also be a key focus point; not just at the onset of app development, but throughout its lifetime.

The impact analysis of bad programming practices, or code smells, in production code has been the focus of numerous studies in software maintenance. Similar to production code, unit tests are also susceptible to bad programming practices which can have a negative impact not only on the quality of the software system but also on maintenance activities. With the present corpus of studies on test smells primarily on traditional applications, there is a need to fill the void in understanding the deviation of testing guidelines in the mobile environment. Furthermore, there is a need to understand the degree to which test smells are prevalent in mobile apps and the impact of such smells on app maintenance. Hence, the purpose of this research is to: (1) extend the existing set of bad test-code practices by introducing new test smells, (2) provide the software engineering community with an open-source test smell detection tool, and (3) perform a large-scale empirical study on test smell occurrence, distribution, and impact on the maintenance of open-source Android apps.

Through multiple experiments, our findings indicate that most Android apps lack an automated verification of their testing mechanisms. As for the apps with existing test suites, they exhibit test smells early on in their lifetime with varying degrees of co-occurrences with different smell types. Our exploration of the relationship between test smells and technical debt proves that test smells are a strong measurement of technical debt. Furthermore, we observed positive correlations between specific smell types and highly changed/buggy test files. Hence, this research demonstrates that test smells can be used as indicators for necessary preventive software maintenance for test suites.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Given the wide availability of Android-based smartphones [2, 1], developers are presented with the opportunity to build mobile apps that can not only reach a vast userbase but also provide users with sophisticated functionality that can tap into hardware features (such as camera, gyroscope, GPS, etc.). This fact is supported by the number of apps currently available on the Google Play Store; as of December 2017, there were roughly 3.5 million apps available on the store [52, 53]. Furthermore, the high-competitiveness of the mobile industry engenders developers to update their apps frequently, and so it becomes challenging for the quality of the software to be maintained.

Quality is a critical driver in all software systems; to such an extent that the success of a system ultimately depends on the quality aspects it exhibits. As such, software project teams utilize a combination of multiple testing strategies [48] during the development and maintenance lifecycle of their software systems. One such popular quality assurance strategy, utilized by software developers, is unit testing. In the object-oriented paradigm, unit testing involves the investigation of every atomic unit in the codebase for quality issues. Through this approach, quality is engraved into the software system much earlier in the development lifecycle and hence positively impacts not only the quality of the system, but also software projects timeline and cost [29]. To this extent, developers are encouraged to write unit tests for the functionality they are implementing in their software systems. With the rise of continuous integration, many project teams have started the integration of unit tests into their build process for real-time visibility of the system quality [9].To further

highlight the importance of unit tests, many of the modern Integrated Development Environments (IDE), such as Android Studio[1], IntelliJ IDEA[2], and Microsoft Visual Studio[3], provide developers with the necessary infrastructure, and framework Application Programming Interfaces (API) to build and evolve test suits in their codebase.

The test code, just like traditional source code, is subject to bad programming practices, also known as anti-patterns, defects and smells [14]. Smells, being symptoms of bad design or implementation decisions, has been proven to be responsible for decreasing the quality of software systems from various aspects, such as making it harder to understand, more complicated to maintain, and more prone to changes and bugs [32] [22] [12] [37]. In this context, several studies on code smells are driven, in general, by optimizing their identification and also proposing more accurate detection strategies [31, 27, 37]. Other studies focused on prioritizing their correction based on their severity in deteriorating the quality of software [55]. Smells also were recently designed as measurements of technical debt [13]. Technical debt is coined as sub-optimal decisions, made by developers, to achieve incomplete but functional goals while saving development effort. Recent studies have been investigating the relationship between technical debt and code smells since both of them negatively impact software quality.

The concept of test smells was initially introduced by van Deursen et al. [57]. Further research in this field has also resulted in the identification of additional test smell types [17], analysis of their evolution and longevity [4] [44], along with patterns to eliminate them [35]. However, as described in Chapter 3, studies around test smells were limited to traditional Java systems. Similarly, there was a growth in research demonstrating how the existence of code smells deteriorate the quality of software designs [5][56]. Although there is a considerable growth of the number of mobile-driven projects, there are no existing studies that analyzed the impact of these bad programming practices on the maintenance of both production and test files, namely their proneness to change and bugs in the mobile

[1] `https://developer.android.com/studio/`
[2] `https://www.jetbrains.com/idea/`
[3] `https://www.visualstudio.com/vs/`

environment. Moreover, several studies have designed strategies on how to detect these smells [50, 8, 16, 40], yet, there is no comprehensive and open-source tool to detect all the types of test smells. As a means of overcoming these challenges, we have extended the set of existing test smell to cover the existing violation of the xUnit testing guidelines [36]. To analyze the lifecycle and impact of these smells, we conducted a large-scale empirical study on JUnit[4]-based unit test suites for 656 open-source Android apps. Further, we defined a series of research questions to support and constrain our investigation to better understand, initially, the existence and distribution of test smells, and more precisely to investigate whether the existence of test smells is an indicator of poor testing quality. Our main findings show that: (1) almost all apps, containing unit tests, had test smells in their test files, introduced in the initial stages of development, their frequency differs per smell type, while their occurrence is similar to traditional Java applications, (2) smells, once introduced into an app tend to remain in the app throughout its lifetime, (3) the existence of test smells acts an indicator of technical debt, and (4) a subset of smell types are found to be more severe in terms of the probability of introducing changes/bugs in test files.

The remainder of this manuscript is as follows: Chapter 2 outlines our motivation for this study along with the research questions we aim to answer, while Chapter 3 enumerates over the related work. Chapter 4 provides the necessary background about existing test smells and our proposed set of test smells. Provided in Chapter 5 are details about the design and methodology of our experiments. In Chapter 6 we answer and discuss the findings of our research questions. We discuss the validity aspects of our study in Chapter 7 and then conclude the paper with Chapter 8 where we summarize our findings and provide insight into our future work.

[4] `https://junit.org`

# Chapter 2

# Research Objective

## 2.1 Motivation

This fast spread of mobile apps has resulted in developers encountering multiple challenges from multiple perspectives including security, privacy, and maintainability. However, the amount of research carried out to support mobile software developers is limited when compared to traditional and web applications. From a smells perspective, past research on Android has been more towards analyzing traditional code smells [42, 30, 18, 46], and Android code smells [21] and its negative impact on maintainability tasks [5, 13, 38, 62]. To the best of our knowledge, an in-depth study of the evolution and severity (change-proneness and bug proneness) of test smells on Android apps has not yet been conducted. Moreover, recent studies have been proving that traditional code smells are being modeled as a possible measurement for technical debt [11, 19]. Whereas, to the best of our knowledge, no study has been conducted to verify whether test smells can be seen as an indicator of test debt. *In this study, we expand on the set of coding best practices for unit tests, and we also aim to understand the current trend of Android developers when it comes to unit testing.* This study will also provide us with insight if Android developers and traditional application developers follow a same/similar pattern with regards to unit testing.

## 2.2 Contribution

Our primary contribution through this study is to enhance the testing experience of developers. We achieve this by:

1. Expanding on the set of existing test smells by proposing additional bad test code practices that negatively impact the quality of the test suite.

2. Building a comprehensive open-source tool, *tsDetect*, to detect bad test code practices.

3. Empirically validating whether smelly test cases can serve as indicators for the deterioration of software quality and so, potentially unwanted system behavior.

*tsDetect* (including source code, documentation, and real-world smell examples) along with the dataset, that was part of our study, is available on our project website.[1]. We welcome and encourage the developer/research community to provide us with feedback and extensions of tsDetect.

## 2.3 Research Questions

We investigate the design of unit tests by studying the occurrence of test smells in Android apps and their impact on the overall quality of the apps through a set of quantitative, comparative and empirical experiments that answer the following research questions:

- **RQ1:** ***How likely are Android apps to contain unit test smells and what characteristics and relationships do the smells exhibit?*** Through multiple experiments, we show the widespread existence of the existing and newly introduced test smells in Android apps, including the distribution and occurrences of each smell type.

[1] `https://testsmells.github.io/`

- **RQ2:** ***To what extent does the severity of test smells increase the risk of test files change- and bug-proneness?*** This question investigates the degree to which, the existence of test smells may lower the maintainability of test suites. We approximate this impact using two dimensions. (1) *Change proneness* - Due to the lack of documentation, lack of using standard JUnit API, redundant test cases, etc. smelly test files tend to be harder to stabilize. (2) *Bug proneness* - Harder to update test cases lead to potential issues related to their inability to identify real faults in the system under test. It can also lead to faults in the test files, i.e., the outcome of the test cases is no longer constrained by the behavior of the system and may be influenced by other factors. This phenomenon is known as the flakiness of test cases, and smells have been proven to provoke it [46]. To evaluate these two dimensions, we perform the analysis of test files revision history along with the issues reported in the projects issue tracker. Then, we report on the probability of each test smell type influencing the increase of bugs and changes to unit test files.

- **RQ3:** ***Do test smells act as an indicator of technical debt within the test suite?*** The intuition behind this research question resides in validating whether the intentional deviation of optimal test files design, due to taking shortcuts in programming, can be in the form of test smells. To address this research question, we observe the co-location of self-admitted technical debt (SATD) in unit test files, with their infection with specific smell types. Thus, we validate whether test smells can act as an indicator of the presence of technical debt in test files.

- **RQ4:** ***What is the degree to which tsDetect can correctly detect test smells?*** Since we are introducing a smell identification tool, it is critical to verify the accuracy of our detection rules. A precision and recall exercise of tsDetect, using qualitative analysis, was utilized to measure the correctness of tsDetect's smell detection ability, for each smell type.

# Chapter 3

# Related Work

Test smells had been initially introduced by van Deursen et al. in the form of 11 unique smells [57]. In contrast with design smells that are defined as violations to Object Oriented design principles, test smells are originated from bad development decisions varying from creating long and hard to maintain test cases, to testing multiple production files using the same test cases. Also, van Deursen et al. [57], found that refactoring test code is different from refactoring production code. Similarly to Fowler's catalog, [14], the authors associated a refactoring strategy for each smell type they reported. These high-level definitions of smells and their refactoring strategies has allowed extensive research to formalize them better and to ease their detection and correction.

Van Rompaey et al. [59] proposed a set of metrics for the detection of two test smells - General Fixtures and Eager Test. They aimed to find out the structural deficiencies encapsulated in a test smell. In [60], the authors extended their approach to demonstrate that metrics can be useful in automating the detection of test smells and confirmed that test smells are related to test design criteria.

Similarly, Reichhart et al. [50] represented test smells using structural metrics in order to construct detection rules by combining metrics with pre-defined thresholds. This approach allowed the automation of the detection of a subset of smell types that had measurable properties.

Breugelmans et al. [8] built a tool, TestQ, which allows developers to visually explore test suites and quantify test smells. TestQ enables developers to inspect the design of a test suite at a high level and helps quickly identify test smell hotspots. Similarly, Koochakzadeh

et al. [23] built a Java plugin for the visualization of redundant tests.

In other studies, Greiler et al. [16] introduced new test smells related to test fixtures - General Fixture, Test Maverick, Dead fields, Lack of cohesion of test methods, Obscure in-line setup and Vague header setup. The researchers also built a detection tool as part of their research.

Neukirchen et al. [40] created T-Rex, a tool that detects any violations of test cases to the Testing and Test Control Notation (TTCN-3) [15]. It serves as an assessment for existing test suites and suggests structural changes to refactor smelly test cases, which does not follow TTCN-3 specifications.

Pinto et al. [47] observed the evolution of test cases over time and have shown that coverage, alone, is not a useful metric to decide about the healthiness of the testing suite, as the system tends to build a natural resistance to the older and unchanged test cases over time. The older the test cases, the lesser chances for them to detect bugs. [7].

Tufano et al. [56] aimed at determining the developer's perception of test smells and came out with results showing that developers could not identify test smells very easily thus resulting in a need for automation. The results also showed that when a test code is committed to the repository that's the time when test smells are usually introduced.

Bavota et al. [5] conducted a human study and proved the strong negative impact of smells on test code understandability and maintainability. Another empirical investigation by the same authors [6] indicated that there is a high diffusion of test smells in both open-source and industrial software systems with 86% of JUnit tests exhibiting at least one test smell. The second study shows that test smells have a strong negative impact on program comprehension and maintenance. These empirical studies highlight the importance for the community to develop tools to detect test smells and automatically refactor them.

Palomba et al. [46] investigated the impact of test smells on flaky test cases, their empirical study measured the distribution of flaky tests among several projects and their collocation with test smell. The experiments have confirmed the negative impact of specific smell types like 'Indirect Testing' and 'Test Run War'.

# Chapter 4

# Test Smells

Test smells are defined as bad programming practices in unit test code (such as how test cases are organized, implemented and interact with each other) that indicate potential design problems in the test source code [58]. Such issues not only have a negative impact on software maintainability but could also have an adverse effect on the testing performance (e.g., flaky tests [46]). In the subsequent subsections, we provide definitions of the unit test smells detected by tsDetect and analyzed in our empirical study. We first provide brief definitions of existing test smells and then provide detailed definitions for the set of proposed test smells. Furthermore, we provide an insight into the tsDetect's detection logic for each smell in Table 4.1. It should be noted that test smells, like traditional code smells, are subjective and open to debate [33]. We welcome both feedback and extensions to the detection logic for the proposed smells.

## 4.1 Literature Test Smells

Provided below is a brief description of literature test smells [57], used in this study.

### 4.1.1 Assertion Roulette

Occurs when a test method has multiple non-documented assertions.

### 4.1.2 Eager Test

Occurs when a test method invokes several methods of the production object.

Table 4.1: Test smell detection rules

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

### 4.1.3 General Fixture

Occurs when a test case fixture is too general, and the test methods only access part of it.

### 4.1.4 Lazy Test

Occurs when multiple test methods invoke the same method of the production object.

### 4.1.5 Mystery Guest

Occurs when a test method utilizes external resources (such as a file or database).

### 4.1.6 Resource Optimism

Occurs when a test method makes an optimistic assumption that the external resource (e.g., File), utilized by the test method, exists

### 4.1.7 Sensitive Equality

Occurs when the `toString` method is used within a test method.

## 4.2 Proposed Test Smells

In this section we extend the existing test smells defined in literature by including a new set of test smells inspired from bad test programming practices mentioned in unit testing based literature ([35, 41, 24, 54]), the JUnit user guide and API, and Android developer documentation[1]. It should be noted that other than for the *Default Test* smell, the set of proposed test smells apply to both traditional Java and Android apps. For these newly introduced test smells, we provide their formal definition, an illustrative example, and our detection mechanism. The examples associated with each test smell were obtained from the dataset that we analyzed in this study. Where possible, we provide the entire code snippet, but in some instances, due to space constraints, we provide only the code statements relevant to the smell. Complete code snippets and detection rules are available on our project website. Furthermore, as elicited later in the experiments, we demonstrate the severity of these

[1] `https://developer.android.com/`

newly introduced smells by showing their impact on change and bug proneness of their infected files.

### 4.2.1 Conditional Test Logic

Test methods need to be simple and execute all statements in the production method. Conditions within the test method will alter the behavior of the test and its expected output, and would lead to situations where the test fails to detect defects in the production method since test statements were not executed as a condition was not met. Furthermore, conditional code within a test method negatively impacts the ease of comprehension by developers. An example is provided in Listing 4.1.

```
/*
    ** Test method contains multiple control statements **
*/
@Test
public void testSpinner() {
    /* ** Control statement #1 ** */
    for (Map.Entry<String, String> entry : sourcesMap.entrySet()) {
        String id = entry.getKey();
        Object resultObject = resultsMap.get(id);
        /* ** Control statement #2 ** */
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            /* ** Control statement #3 ** */
            if (result.testSpinner.runTest) {
                System.out.println("Testing " + id + " (testSpinner)");
                AnswerObject answer = new AnswerObject(entry.getValue(), "", new CookieManager(), "");
                EventsScraper scraper = new EventsScraper(RuntimeEnvironment.application, answer);
                SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
                assertEquals(spinnerAdapter.getCount(), result.testSpinner.data.size());
                /* ** Control statement #4 ** */
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.data.get(i));
                }
            }
        }
    }
}
```

Listing 4.1: Example - Conditional Test Logic.

### 4.2.2 Constructor Initialization

Ideally, the test suite should not have a constructor. Initialization of fields should be in the `setUp()` method. Developers who are unaware of the purpose of `setUp()` method would give rise to this smell by defining a constructor for the test suite. An example is provided in Listing 4.2.

```
public class TagEncodingTest extends BrambleTestCase {
    private final CryptoComponent crypto;
    private final SecretKey tagKey;
    private final long streamNumber = 1234567890;
    /*
        ** Constructor initializing field variable **
    */
    public TagEncodingTest() {
        crypto = new CryptoComponentImpl(new TestSecureRandomProvider());
        tagKey = TestUtils.getSecretKey();
    }
    @Test
    public void testKeyAffectsTag() throws Exception {
        Set<Bytes> set = new HashSet<Bytes>();
        for (int i = 0; i < 100; i++) {
            byte[] tag = new byte[TAG_LENGTH];
            SecretKey tagKey = TestUtils.getSecretKey();
            /*
                ** Field variable utilized in test method **
            */
            crypto.encodeTag(tag, tagKey, PROTOCOL_VERSION, streamNumber);
            assertTrue(set.add(new Bytes(tag)));
        }
    }
    .....
}
```

Listing 4.2: Example - Constructor Initialization.

### 4.2.3 Default Test

By default Android Studio creates default test classes when a project is created. These classes are meant to serve as an example for developers when writing unit tests and should either be removed or renamed. Having such files in the project will cause developers to start adding test methods into these files, making the default test class a container of all test cases. This also would possibly cause problems when the classes need to be renamed in the future. An example is provided in Listing 4.3.

```
/*
    ** Default test class created by Android Studio **
*/
public class ExampleUnitTest {
    /*
        ** Default test method created by Android Studio **
    */
    @Test
    public void addition_isCorrect() throws Exception {
        assertEquals(4, 2 + 2);
    }

    /*
        ** Actual test method **
    */
    @Test
    public void shareProblem() throws InterruptedException {
        .....
        Observable.just(200)
        .subscribeOn(Schedulers.newThread())
        .subscribe(begin.asAction());
        begin.set(200);
        Thread.sleep(1000);
        assertEquals(beginTime.get(), "200");
        .....
    }
    .....
}
```

Listing 4.3: Example - Default Test.

### 4.2.4 Duplicate Assert

This smell occurs when a test method tests for the same condition multiple times within the same test method. If the test method needs to test the same condition using different values, a new test method should be utilized; the name of the test method should be an indication of the test being performed. Possible situations that would give rise to this smell include: (1) developers grouping multiple conditions to test a single method; (2) developers performing debugging activities; and (3) an accidental copy-paste of code. An example is provided in Listing 4.4.

```
@Test
public void testXmlSanitizer() {
    .....
    valid = XmlSanitizer.isValid("Fritz-box");
    /*
        ** Assert statements are the same **
    */
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    /*
        ** Assert statements are the same **
    */
    assertEquals("Minus is valid", true, valid);
       System.out.println("Minus test - passed");
    .....
}
```

Listing 4.4: Example - Duplicate Assert.

### 4.2.5 Empty Test

Occurs when a test method does not contain executable statements. Such methods are possibly created for debugging purposes and then forgotten about or contains commented out code. An empty test can be considered problematic and more dangerous than not having a test case at all since JUnit will indicate that the test passes even if there are no executable statements present in the method body. As such, developers introducing behavior-breaking changes into production class, will not be notified of the alternated outcomes as JUnit will report the test as passing. An example is provided in Listing 4.5.

```
/*
    ** Test method without executable statements **
*/
public void testCredGetFullSampleV1() throws Throwable{
//  ScrapedCredentials credentials =  innerCredTest(FULL_SAMPLE_v1);
//  assertEquals("p4ssw0rd", credentials.pass);
//  assertEquals("user@example.com",credentials.user);
}
```

Listing 4.5: Example - Empty Test.

### 4.2.6 Exception Handling

This smell occurs when the passing or failing of a test method is explicitly dependent on the production method throwing an exception. Developers should utilize JUnit's exception handling features to automatically pass/fail the test instead of writing custom exception handling code or throwing an exception. An example is provided in Listing 4.6.

```
@Test
public void realCase() {
    .....
    a.getMeasures().add(new Measure(p47, 281.3521, 100.0471, 108.384, 1.63));
    /*
        ** Fails the test when an exception occurs **
    */
    try {
        a.compute();
    } catch (CalculationException e) {
    Assert.fail(e.getMessage());
    }
    Assert.assertEquals("233.2405", this.df4.format(a.getResults().get(0).getUnknownOrientation()));
    .....
}
```

Listing 4.6: Example - Exception Handling.

### 4.2.7 Ignored Test

JUnit 4 provides developers with the ability to suppress test methods from running. However, these ignored test methods result in overhead since they add unnecessary overhead with regards to compilation time, and increases code complexity and comprehension. An example is provided in Listing 4.7.

```
@Test
/*
    ** This test will not be executed due to the @Ignore annotation **
*/
@Ignore("disabled for now as this test is too flaky")
public void peerPriority() throws Exception {
    final List<InetSocketAddress> addresses = Lists.newArrayList(
        new InetSocketAddress("localhost", 2000),
        new InetSocketAddress("localhost", 2001),
        new InetSocketAddress("localhost", 2002)
    );
    peerGroup.addConnectedEventListener(connectedListener);
    .....
}
```

Listing 4.7: Example - Ignored Test.

### 4.2.8 Magic Number Test

Occurs when assert statements in a test method contain numeric literals (i.e., magic numbers) as parameters. Magic numbers do not indicate the meaning/purpose of the number. Hence, they should be replaced with constants or variables, thereby providing a descriptive name for the input. An example is provided in Listing 4.8.

```
@Test
public void testGetLocalTimeAsCalendar() {
    Calendar localTime = calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D),     Calendar.getInstance());
    /*
        ** Numeric literals are used within the assertion statement **
    */
    assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
    assertEquals(30, localTime.get(Calendar.MINUTE));
}
```

Listing 4.8: Example - Magic Number Test.

### 4.2.9 Redundant Print

Print statements in unit tests are redundant as unit tests are executed as part of an automated process with little to no human intervention. Print statements are possibly used by developers for traceability and debugging purposes and then forgotten. An example is provided in Listing 4.9.

```
@Test
public void testTransform10mNEUAndBack() {
    Leg northEastAndUp10M = new Leg(10, 45, 45);
    Coord3D result = transformer.transform(Coord3D.ORIGIN, northEastAndUp10M);
    /*
        ** Print statement does not serve any purpose **
    */
    System.out.println("result = " + result);
    Leg reverse = new Leg(10, 225, -45);
    result = transformer.transform(result, reverse);
    assertEquals(Coord3D.ORIGIN, result);
}
```

Listing 4.9: Example - Redundant Print.

### 4.2.10 Redundant Assertion

This smell occurs when test methods contain assertion statements that are either always true or always false. Developers introduce this smell for debugging purposes and then forget to remove it. Listing 4.10 provides an example.

```
@Test
public void testTrue() {
    /*
        ** Assert statement will always return true **
    */
    assertEquals(true, true);
}
```

Listing 4.10: Example - Redundant Assertion.

### 4.2.11 Sleepy Test

Explicitly causing a thread to sleep can lead to unexpected results as the processing time for a task can differ on different devices. Developers introduce this smell when they need to pause execution of statements in a test method for a certain duration (i.e., simulate an external event) and then continuing with execution. An example is provided in Listing 4.11.

```
public void testEdictExternSearch() throws Exception {
    .....
    DictEntry entry = (DictEntry) lv.getItemAtPosition(0);
    assertEquals("Searching", entry.english);
    /*
        ** Forcing the thread to sleep **
    */
    Thread.sleep(500);
    final Intent i2 = getStartedActivityIntent();
    .....
}
```

Listing 4.11: Example - Sleepy Test.

### 4.2.12 Unknown Test

An assertion statement is used to declare an expected boolean condition for a test method. By examining the assertion statement, it is possible to understand the purpose of the test method. However, It is possible for a test method to be written sans an assertion statement, in such an instance JUnit will show the test method as passing if the statements within the test method did not result in an exception when executed. New developers to the project will find it difficult in understanding the purpose of such test methods (more so if the name of the test method is not descriptive enough). An example is provided in Listing 4.12.

```
/*
    ** Test method without an assertion statement **
    ** Test method name is not descriptive enough to understand its purpose **
*/
@Test
public void hitGetPOICategoriesApi() throws Exception {
    POICategories poiCategories = apiClient.getPOICategories(16);
    for (POICategory category : poiCategories) {
      System.out.println(category.name() + ": " + category);
    }
}
```

Listing 4.12: Example - Unknown Test.

## 4.3 tsDetect

To provide developers with a mechanism to detect, both the existing set test smells and the proposed set of test smells, we implemented an open-source tool for developers to run against their unit test files. tsDetect is available as a standalone jar file and requires a list of file paths as input, and will automatically scan the provided list of files for the occurrences of all test smell types. Internally, tsDetect utilizes JavaParser [2] to parse the Java source file through the use of an abstract syntax tree (AST). Depending on the type of smell being detected, we override the appropriate `visit()` method to perform our analysis/detection. The design of tsDetect is such that it facilitates the inclusion of detection rules for additional smell types. The output from tsDetect is in the form of a CSV file. Each row in the CSV file corresponds to a unit test file, and the associated columns contain boolean values indicating if the specific smell type is present or not. Refer Appendix A for UML class diagrams.

[2] `https://javaparser.org/`

# Chapter 5

# Methodology

To answer our research questions, we conducted a two-phased approach that consists of: (1) data mining and (2) smell detection. The Data Mining Phase consists of collecting datasets from multiple sources[1] while the Detection Phase involved the analysis of the collected datasets for the existence of test smells, along with the impact of such smells on multiple project traits. Due to performance requirements associated with this volume of data mining, smell detection, and data analysis, the activities associated with both phases were performed on dedicated virtual machines with 2 CPU's, 16 GB of RAM and over 2 TB of hard disk space. The details of each phase are described in the following subsections.

## 5.1 Data Mining Phase

Similar to prior research [26, 25, 3], for this study we utilized F-Droid's[2] index of open-source Android apps. From the 2,596 repositories listed on F-Droid (at the commencement of this study), we narrowed our selection to only repositories hosted in publicly accessible Git-based version control systems. Our dataset only consisted of repositories that were not duplicated/forked; we did this by ensuring that the source URL's and commit SHA's were unique. For each of the cloned repositories, we retrieved: (1) the entire commit log[3], (2) list

[1] The data mining activities occurred during December 2017 and took around two weeks

[2] `https://f-droid.org/`

[3] App history commits, based on the first and most recent commit of the 'AndroidManfiest.xml' file, range from February 2008 to December 20017

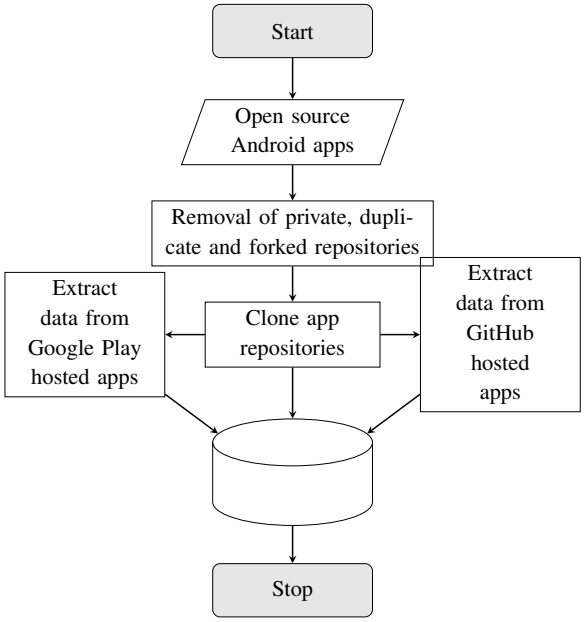


Figure 5.1: Overview of the Data Mining Phase

Table 5.1: Overview of data obtained in the Data Mining Phase

| **Item** | **Value** |
|---|---|
| Total cloned repositories | 2,011 |
| Cloned apps available on Google Play | 1,222 |
| Cloned apps hosted on GitHub | 1,835 |
| Cloned apps utilizing GitHub's issue tracker | 808 |
| Total number of commit log entries | 1,037,236 |
| Total number of Java files affected by commits | 6,379,006 |
| Total volume of repositories cloned | 53.8 GB |
| Total volume of test files collected | 3.63 GB |

of all files affected by each commit, (3) all available tags, and (4) the complete version history of all identified test files and their corresponding production files. Further, for projects hosted on GitHub[4], we retrieved popularity metrics (including the number of Stargazers, Forks, Subscribers, and Releases) and issue tracker details associated with each project.

[4] `https://github.com/`

Finally, we crawled the Google Play Store[5] to retrieve the Review Score associated with the cloned apps.Depicted in Figure 5.1 is an overview of the Data Mining Phase process, while Table 5.1 provides an overview of the data collected.

## 5.2 Detection Phase

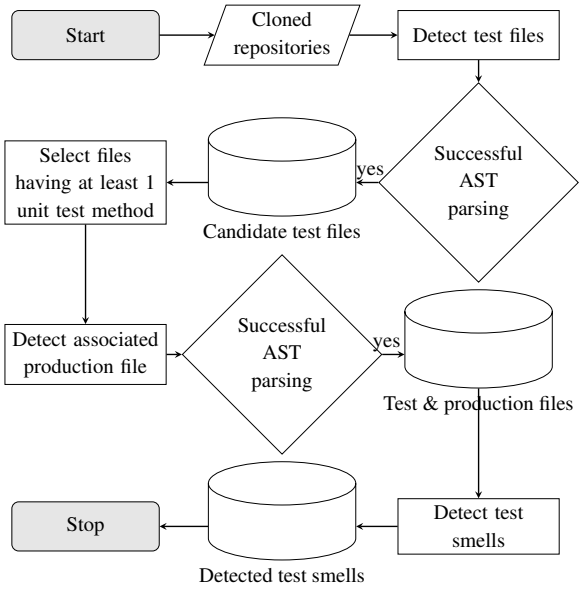


Figure 5.2: Overview of the Detection Phase

The primary purpose of this phase is to detect test smells occurring in unit test files. However, as this is an empirical study, prior to the detection of test smells, we perform additional activities to detect: (1) unit test files, contained in the apps project repository, and (2) the production files associated with the detected test files. As depicted in Figure 5.2, we first identify candidate unit test files, which exist throughout the lifetime of the app. Next, we identify the production files associated with the detected test files, and finally, feed in the list of identified test and production files into tsDetect. In the subsequent subsections, we describe in detail each of the detection activities. An overview of the data collected/analyzed in this phase is provided in Table 5.2 and 5.3.

[5] `https://play.google.com/store/`

Table 5.2: Overview of data obtained in the Detection Phase

| **Item** | **Value** |
|---|---|
| Apps containing test files | 656 |
| Candidate test files detected | 206,598 |
| Test files with an associated production file | 112,514 |
| Test methods analyzed | 1,187,055 |
| Test files associated with a GitHub issue | 5,693 |
| ***Test smells*** | |
| Test files not exhibiting any test smells | 5,915 |
| Test files containing 1 or more smells | 175,866 |
| Test files containing only 1 type of smell | 22,927 |
| Test files containing 2 to 5 types of smells | 95,565 |
| Test files containing 6 to 10 types of smells | 33,898 |
| Test files containing over 10 types smells | 3,317 |

Table 5.3: Statistical summary of test file and smell occurrence

| **Item** | **Min.** | **1st Qu.** | **Median** | **Mean** | **3rd Qu.** | **Max.** |
|---|---|---|---|---|---|---|
| Test Files Per App | 1 | 1 | 2 | 17.75 | 10 | 510 |
| Smells Per File | 0 | 2 | 3 | 3 | 5 | 13 |

### 5.2.1 Test File Detection

Ideally, when writing JUnit test cases, developers should follow a naming convention when creating unit test files. The recommended naming conventions [6] consists of either prepending or appending the word ‘Test‘ to the name of the production file that is to be tested (i.e., Test*.java and *Test.java). For example, if the file ‘Example.java’ contains test methods, then it should be named ‘TestExample.java’ or ‘ExampleTest.java’.

Following the above naming recommendations, we utilized a tool-based automation approach in identifying unit test files. First, our tool identified all ‘.java’ files where the

[6] `http://junit.org/junit4/faq.html#running_15`

filename either started or ended with the word ‘Test’. Next, for each of the identified Java source files our the tool utilized JavaParser to obtain an AST from the file. The use of AST’s in software repository mining and analysis studies is a technique that is frequently utilized by researchers ( [51, 34]). The purpose of using the AST was twofold; firstly, we were able to eliminate Java files that contain syntax errors and secondly, we were able to detect if the file contained JUnit-based unit test methods accurately. For a file to contain a unit test method, the method should either have an annotation called ‘@Test’ (JUnit 4) or the method name should start with ‘test’ (JUnit 3). The result of this activity resulted in a set of candidate unit test files. From this resultset, we only considered unit test files that had one or more unit test methods as valid unit test files.

### 5.2.2 Production File Detection

Our purpose for identifying the production file, associated with a unit test file, was to detect if the unit test file contains an *Eager Test* and/or *Lazy Test* smell. Ideally, mappings between test and production files would be contained in an oracle. However, due to the vast quantity of apps in our study, a manual construction of an oracle is not feasible. Hence, we had to automate the mapping process. First, for each unit test file, we identified its associated production file(s) by searching the apps repository tree for a file that has the same name as the test file, but without the word ‘Test’. Next, for each of the identified production files, we utilized JavaParser to obtain an AST from the file to ensure that the file is syntactically correct. Finally, we utilized the results obtained from this activity as input for tsDetect. As a means of quality assurance, we ran SQLite’s `Random()` function on our resultset to select 50 random pairing of test and production files as a means of verifying the mapping process; to the best of our knowledge, all 50 pairings were deemed valid.

### 5.2.3 Test Smell Detection

After the identification of all unit test files and their associated production files, we ran tsDetect to detect the occurrence and distribution of test smells in the unit test files. Utilizing

JavaParser, tsDetect parses the Java source file and builds an AST for the same. Depending on the type of smell being detected, we override the appropriate `visit()` method to perform our analysis/detection. Results provided by tsDetect were saved in a database for analysis/interpretation.

## 5.3 Traditional Java Applications

Table 5.4: Volume and popularity traits of the mined Java apps

| **Item** | **Value** |
|---|---|
| ***Volume traits*** | |
| Test files with 1 or more smells | 82,261 |
| Test methods analyzed | 634,877 |
| Test files associated with a GitHub issue | 103,608 |
| ***Popularity traits*** | |
| Average number of stargazers per app | 4,242 |
| Average number of commits per app | 3,214 |
| Average number of forks per app | 1,773 |
| Average number of subscribers per app | 467 |
| Average number of contributors per app | 103 |

Even though Android apps are the primary focus of this study, we mine traditional Java applications as a means of performing a comparison on the distribution of test smells between these environments. To this extent, we mined the GitHub repositories of 18 popular open-source Java applications. To ensure a recent, yet active and sizeable dataset key selection criteria for the Java applications included: (1) the utilization of GitHub to track and manage issues with a minimum of 100 closed, non-pull request based issues, (2) utilization of JUnit as the testing framework, (3) a repository age of not less than five years, and (4) over 750 commits during the lifetime of the application. We utilized GitHub and Open

Hub[7] to manually search for projects that satisfy the key requirements mentioned above. We ceased our search once we had enough applications that provided use with roughly 50% more smelly test files than our Android dataset. Table 5.4 provides an overview of some volume and popularity traits of the Java applications. Our approach to mining and analyzing the Java applications was similar to our Android mining and analysis approach. We first mined each app to retrieve all revisions of the apps' unit test files (and their associated production files). From these mined source code files, we identified the test smells that occur in the lifetime of the app by analyzing the files with tsDetect.

[7] https://www.openhub.net/

# Chapter 6

# Analysis & Discussion

In this chapter, we present answers to our research questions by analyzing the occurrence and impact of test smells in the studied apps.

## 6.1 RQ1: How likely are Android apps to contain unit test smells, and what characteristics and relationships do the smells exhibit?

We address RQ1 through a series of sub-RQ's, related to various aspects of test smells such as their existence, evolution, co-occurrence, and distribution among traditional and mobile software systems. By running tsDetect on the version history of all unit test files (identified by enumerating over the app's git commit log), we were able to obtain the history of test smells occurring during the lifetime of the app. We then utilized this data in the following sub-RQ's when formulating our analysis.

### 6.1.1 RQ1.1: Are Android apps, that contain a test suite, prone to test smells?

Out of the 656 apps, which contained unit tests, only 21 apps (approximately 3%) did not exhibit any test smells. Analyzing the smell free apps, we observed that the non-smelly apps contained significantly less unit test files, within the lifetime of the app, than the smelly apps. The low count of unit test files in the non-smelly apps cannot be attributed to the size of the project as the count of Java files occurring in the lifetime of smelly and

Table 6.1: Statistical summary of the distribution of source code files in smelly and non-smelly apps

| Item | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|---|---|---|---|---|---|---|
| Non-Smelly Apps - Distinct Test Files | 1 | 1 | 1 | 1.1 | 1 | 2 |
| Non-Smelly Apps - Distinct Java Files | 37 | 154 | 183 | 332.1 | 276 | 1255 |
| Smelly Apps - Distinct Test Files | 1 | 1 | 3 | 18.3 | 10 | 510 |
| Smelly Apps - Distinct Java Files | 1 | 28.5 | 106 | 325 | 330 | 5780 |

non-smelly apps was similar. Hence, a possible explanation for the absence of the test smells in the 21 apps can be due to low unit testing coverage in the app. Table 6.1 reports on the statistics of the distribution of production test and source code files in smelly and non-smelly apps.

A typical train of thought concerning smells is that as the test suite of an app increases so does the occurrences of smells; due to the addition of more test methods (i.e., test cases) to exercise new production code. We verify this claim via a hypothetical null test; where we define the following null hypothesis:

**Null Hypothesis 1** *The existence of unit test smells, in an app, does not change as functionalities of the app continues to grow over time.*

Based on a Shapiro-Wilk Normality Test on our dataset of unit test file and test smell occurrence, we observed that the dataset is of a non-normal distribution. To this extent, we performed a Spearman rank correlation coefficient test to assess the association between the volume of test smells and test files occurring throughout the history of the apps. As shown in Table 6.2, not surprisingly, we obtained a strong positive and statistically significant correlation between the two variables. Therefore, we can reject the Null Hypothesis 1 and statistically confirm that test smells exhibited by an app increase as the unit test files in the app increase.

Table 6.2: Spearman's correlation calculation on test smells and test files. Bold indicates that the obtained result was statistically significant (p$<$ 0.05).

| **Variable #1** | **Variable #2** | **Correlation ($\rho$)** |
|---|---|---|
| Total Test Smells | Total Test Files | **0.90** |

Table 6.3: Correlation analysis between test smells and app popularity traits. Bold indicates that the obtained result was statistically significant (p$<$ 0.05).

| **Variable #1** | **Variable #2** | **Correlation ($\rho$)** |
|---|---|---|
| Total Test Smells | Total Authors | **0.38** |
| | Total Forks | **0.33** |
| | Total Tags | **0.43** |
| | Total Releases | **0.23** |
| | Total Stargazers | **0.35** |
| | Total Subscribers | **0.35** |
| | Total Google Play Reviewers | **0.28** |
| | Google Play Review Score | 0.08 |

We further extended the study on test smell occurrence by investigating the degree of correlation between test smells and certain project popularity traits. An overview of our findings is provided in Table 6.3. Even though we obtained statistically significant correlation values, the correlations were in the positive weak ($\rho \geq 0.3$) to moderate ($\rho \leq 0.5$) range. The count of Authors, Forks, Stargazers, and Subscribers associated with a repository indicate the popularity of the project [20]. This positive correlation acts as an indicator for app developers to ensure that their test suites are in a state that requires less maintenance effort. Developers need to ensure that the test code not only provides the maximum possible coverage but is also comprehensible (self-documentable). Failure by developers to easily and quickly understand the test suite codebase results in more time and errors in maintenance activities, which in turn would have negative consequences on the popularity of the app. Interestingly, we observed the absences of a correlation between an

apps test smells and its Google Play Review Score. This phenomenon might not necessarily be due to lack of test smells; one possible reason could be lack of test coverage.

The results we presented on app popularity and test smells are exploratory and require further granular research. Popularity can be subjective, and future research should complement the quantitative results we presented with qualitative data.

Table 6.4: Volume of apps and files exhibiting each smell type

| **Smell Type** | **Smell Exhibition In** | |
|---|---|---|
| | **Apps** | **Files** |
| Assertion Roulette | 52.28% | 58.46% |
| Conditional Test Logic | 37.32% | 28.67% |
| Constructor Initialization | 20.47% | 11.70% |
| Default Test | 42.20% | 0.32% |
| Duplicate Assert | 31.81% | 31.33% |
| Eager Test | 42.99% | 38.68% |
| Empty Test | 16.38% | 1.08% |
| Exception Handling | 84.57% | 49.18% |
| General Fixture | 25.51% | 11.67% |
| Ignored Test | 15.28% | 3.00% |
| Lazy Test | 39.06% | 29.50% |
| Magic Number Test | 77.01% | 34.84% |
| Mystery Guest | 36.38% | 11.65% |
| Redundant Assertion | 12.91% | 3.87% |
| Redundant Print | 14.02% | 0.92% |
| Resource Optimism | 15.75% | 9.79% |
| Sensitive Equality | 21.10% | 9.19% |
| Sleepy Test | 12.60% | 2.04% |
| Unknown Test | 47.09% | 34.38% |

**RQ1.2: What is the frequency and distribution of test smells in Android apps over time?**

To further aid our discussion on the occurrence of test smells, in the analyzed apps, we calculated the distribution of each test smell type from the total quantity of detected test

Table 6.5: Co-occurrence of test smells

| Smell Type | ASR | CTL | CNI | DFT | EMT | EXP | GFX | MGT | RPR | RAS | SEQ | SLT | EGT | DAS | LZT | UKT | IGT | ROP | MNT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ASR | | 31% | 9% | 0% | 1% | 49% | 13% | 13% | 1% | 3% | 11% | 2% | 54% | 46% | 37% | 23% | 3% | 13% | 52% |
| CTL | 62% | | 18% | 0% | 2% | 58% | 14% | 25% | 2% | 7% | 9% | 5% | 44% | 39% | 33% | 46% | 6% | 20% | 40% |
| CNI | 43% | 44% | | 0% | 1% | 84% | 12% | 22% | 1% | 3% | 3% | 6% | 32% | 24% | 24% | 57% | 2% | 12% | 18% |
| DFT | 0% | 0% | 0% | | 1% | 99% | 0% | 23% | 1% | 0% | 0% | 0% | 0% | 0% | 0% | 2% | 0% | 0% | 76% |
| EMT | 69% | 45% | 10% | 0% | | 42% | 28% | 8% | 0% | 0% | 4% | 1% | 35% | 32% | 18% | 100% | 2% | 2% | 47% |
| EXP | 58% | 34% | 20% | 1% | 1% | | 15% | 19% | 1% | 5% | 6% | 4% | 35% | 32% | 32% | 40% | 3% | 18% | 39% |
| GFX | 66% | 35% | 12% | 0% | 3% | 63% | | 10% | 1% | 1% | 10% | 3% | 49% | 42% | 47% | 43% | 3% | 8% | 38% |
| MGT | 67% | 61% | 22% | 1% | 1% | 79% | 10% | | 1% | 3% | 5% | 4% | 42% | 40% | 29% | 46% | 2% | 63% | 42% |
| RPR | 46% | 74% | 7% | 0% | 1% | 46% | 19% | 6% | | 1% | 9% | 1% | 25% | 22% | 21% | 61% | 2% | 5% | 32% |
| RAS | 45% | 50% | 8% | 0% | 0% | 70% | 4% | 10% | 0% | | 2% | 3% | 46% | 14% | 40% | 4% | 8% | 7% | 40% |
| SEQ | 71% | 28% | 4% | 0% | 0% | 34% | 13% | 6% | 1% | 1% | | 2% | 48% | 44% | 35% | 20% | 3% | 3% | 52% |
| SLT | 60% | 67% | 36% | 0% | 0% | 100% | 18% | 20% | 0% | 5% | 9% | | 48% | 38% | 31% | 53% | 5% | 14% | 26% |
| EGT | 82% | 33% | 10% | 0% | 1% | 45% | 15% | 13% | 1% | 5% | 11% | 3% | | 46% | 61% | 19% | 1% | 11% | 49% |
| DAS | 86% | 36% | 9% | 0% | 1% | 51% | 16% | 15% | 1% | 2% | 13% | 2% | 57% | | 44% | 26% | 3% | 13% | 60% |
| LZT | 72% | 32% | 10% | 0% | 1% | 53% | 19% | 11% | 1% | 5% | 11% | 2% | 79% | 47% | | 26% | 1% | 9% | 47% |
| UKT | 39% | 38% | 19% | 0% | 3% | 57% | 15% | 16% | 2% | 0% | 5% | 3% | 21% | 24% | 22% | | 7% | 14% | 25% |
| IGT | 50% | 53% | 7% | 0% | 1% | 49% | 10% | 6% | 1% | 10% | 8% | 3% | 19% | 32% | 13% | 75% | | 6% | 35% |
| ROP | 77% | 60% | 15% | 0% | 0% | 92% | 10% | 75% | 0% | 3% | 3% | 3% | 44% | 41% | 26% | 48% | 2% | | 45% |
| MNT | 88% | 33% | 6% | 1% | 1% | 55% | 13% | 14% | 1% | 4% | 14% | 2% | 55% | 54% | 40% | 25% | 3% | 13% | |

*Abbreviations:*
ASR = Assertion Roulette | CTL = Conditional Test Logic | CNI = Constructor Initialization | DFT = Default Test | EMT = Empty Test | EXP = Exception Handling |
GFX = General Fixture | MGT = Mystery Guest | RPR = Redundant Print | RAS = Redundant Assertion | SEQ = Sensitive Equality | SLT = Sleepy Test | EGT = Eager Test |
DAS = Duplicate Assert | LZT = Lazy Test | UKT = Unknown Test | IGT = Ignored Test | ROP = Resource Optimism | MNT = Magic Number Test |

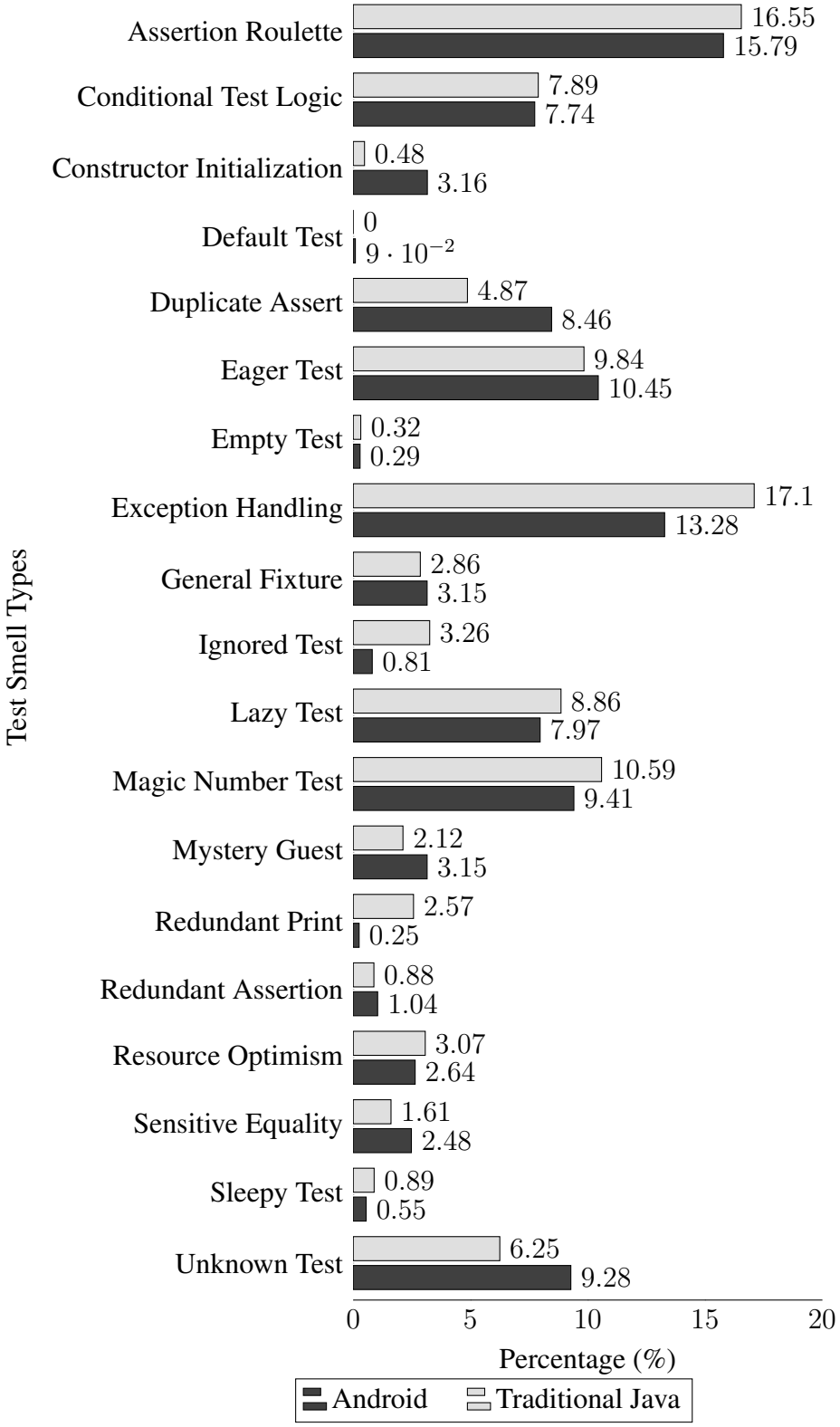


Figure 6.1: Distribution of test smells throughout the lifetime of traditional Java and Android apps

smells (Figure 6.1), the volume of apps and unit test files that exhibit each smell type (Table 6.4), and the co-occurrence of test smells (Table 6.5). We observed that the smell *Assertion Roulette* occurred the most when compared to the other smells. Further, we also observed that this smell also occurred in over approximately 50% of the analyzed apps and unit test files. As claimed by [49], a reason for the high occurrence of the *Assertion Roulette* this could be due to developers verifying the testing environment prior to the behavior of the testing class. The high occurrence of the *Exception Handling* smell could be attributed to developers using IDE productivity tools to auto-generate the skeleton test method. For example, IntelliJ IDEA provides the ability to auto-generate the skeleton for test methods based on a pre-defined template. As such, developers might be utilizing templates in which the test method throws a general exception. Since an *Eager Test* smell is attributed to a test method exercising multiple production methods, a high occurrence of this smell can also be due to developers either testing the environment or initiating/setting-up the object under test. This phenomenon is further evident by the high co-occurrence (over 80%) of the *Eager Test* smell with the *Assertion Roulette* smell. Another smell with a high distribution is the *Magic Number Test* smell. Typically, test methods utilize assertion statements to compare the expected result returned by a production method against the actual value; therefore justifying the high occurrence of this smell. Furthermore, it also shows that developers tend to favor using numerical literals as parameters in the assertion methods. Further evidence of this is the high co-occurrence of this smell with the smell *Assertion Roulette* (approximately 88%).

Interestingly, the smell *Unknown Test* shows a moderate-to-high value in the distribution of smells and occurs in nearly half of the analyzed apps. This means that developers tend to write unit test methods without an assertion statement or utilizing JUnit's exception handling features. However, we noticed that this smell has a high co-occurrence (over 55%) with the smell Exception Handling; a possible reason for this event is that developers determine the passing/failing of a test method based on the exception thrown by the called production method. The other smells that show a moderate distribution include *Duplicate*

*Assertion*, *Lazy Test*, and the *Conditional Test Logic* smells. These three smells also occur in less than half of the analyzed apps.

The remainder of the detected smells have a low distribution. We observed that the *Mystery Guest* and *Resource Optimism* smell have a similar distribution occurrence and also share a similar co-occurrence with each other. This means that even though developers do not frequently utilize external resources, they tend to assume that the external resource exists when they do consume the resource. Not surprisingly, the *Default Test* smell has an exceptionally high co-occurrence with the *Exception Handling* and *Magic Number Test* smells. This phenomenon can be explained by examining the default unit test files automatically added by Android Studio; the default file contains a single exemplar test method that contains an assertion method with numeric literals as parameters and throws a default exception. However, the minor co-occurrences with other smells imply that developers also tend to update the default files with custom test cases. Even though the distribution of the *Redundant Print* smell is low, it has a high co-occurrence with the *Conditional Test Logic* smell. A possible reason for this behavior can be attributed to developers utilizing the print methods for debugging purposes when building/evaluating the conditional statements contained in the test methods.

### 6.1.2 RQ1.3: Do specific test smells have an impact on test suite characteristics?

To further understand the degree to which test smells can impact software maintenance activities of an Android app, we studied the impact of each test smell type on specific characteristics of the apps test suite. Our investigation followed a similar approach to [43].

First, for each smell type, we obtained the apps that exhibited the smell. Next, for each smelly app, we obtained the total number of JUnit classes contained in the app. From these classes, we obtained the total number of methods and lines of code (LOC). Additionally, we also obtained the count of co-located smells exhibited by the class for each smell type. Finally, we computed the correlation between the number of instances of the specific smell

and the derived characteristics.To facilitate our investigation, we defined the following Null Hypotheses:

**Null Hypothesis 2** *The existence of a specific test smell does not have an impact on an app's test suite characteristics.*

Table 6.6: Correlation analysis between test smell instances and system characteristics

| Smell Type | Correlation ($\rho$) with | | | |
|---|---|---|---|---|
| | JUnit Classes | Methods | Lines of Code | Co-Located Smells |
| Assertion Roulette | **0.84** | **0.85** | **0.87** | **0.80** |
| Conditional Test Logic | **0.73** | **0.73** | **0.74** | **0.73** |
| Constructor Initialization | **0.48** | **0.48** | **0.47** | **0.45** |
| Default Test | **-0.38** | **-0.52** | **-0.56** | **-0.35** |
| Duplicate Assert | **0.71** | **0.72** | **0.74** | **0.71** |
| Eager Test | **0.79** | **0.80** | **0.80** | **0.79** |
| Empty Test | **0.46** | **0.46** | **0.43** | **0.44** |
| Exception Handling | **0.79** | **0.70** | **0.67** | **0.81** |
| General Fixture | **0.66** | **0.66** | **0.65** | **0.64** |
| Ignored Test | **0.46** | **0.47** | **0.46** | **0.42** |
| Lazy Test | **0.75** | **0.77** | **0.77** | **0.75** |
| Magic Number Test | **0.63** | **0.56** | **0.54** | **0.66** |
| Mystery Guest | **0.53** | **0.48** | **0.49** | **0.54** |
| Redundant Assertion | **0.39** | **0.39** | **0.38** | **0.37** |
| Redundant Print | **0.38** | **0.37** | **0.38** | **0.39** |
| Resource Optimism | **0.49** | **0.50** | **0.50** | **0.51** |
| Sensitive Equality | **0.56** | **0.58** | **0.58** | **0.56** |
| Sleepy Test | **0.44** | **0.43** | **0.43** | **0.44** |
| Unknown Test | **0.79** | **0.81** | **0.80** | **0.73** |

Table 6.6 provides the result of computing the Spearman rank correlation coefficient for each characteristic. Not surprisingly the majority of the smell types yielded positive and statistically significant (i.e., $p < 0.05$) values. The Default Test smell is the only smell that exhibits a weak negative correlation. This is not surprising since this smell is associated with the sample unit test file included by Android Studio and does exercise code in production files. All other smells exhibit positive weak ($0.3 \leq \rho < 0.5$) to strong ($0.5 \leq \rho < 1.0$) correlations [28] with test suite characteristics. Given these results Null Hypothesis 2 can

be rejected. As such, developers need to pay careful attention to the quality of the code that they write for unit tests, much in the same way they do for production tests, as a means to reduce the effort required to maintain the test code.

### 6.1.3 RQ1.4: How do test smells exhibited by Android apps compare against traditional Java applications?

Most of the prior research in this area has focused on test smells exhibited by traditional Java applications. As part of this study, we performed a comparison of test smells occurring in Android and traditional Java applications to understand the degree to which the distribution of test smells in these environments differ. From Figure 6.1, it is observed that both environments contain a high distribution of the *Exception Handling*, *Assertion Roulette*, *Magic Number Test* and *Eager Test* smells. For most of the other test smells, we noticed that the difference in occurrence is minor. Given that native Android apps are Java-based and also utilize the same JUnit framework along with best practices, the similarity in the distribution of test smells in both environments is not surprising. Furthermore, when compared to past research [6, 45] we observed that our findings, for the common set of test smells, are also similar.

### 6.1.4 RQ1.5: When are test smells first introduced into the project?

Our study on the introduction of test smells into a project involved the analysis of commits to identify when the first commit of a smelly test file occurs and the number of smells introduced when a unit test file is added to the project.

For each app in our study, we identified the very first instance of a smelly unit test file and then identified when this file was introduced (i.e., committed) into the apps' project repository. Given the vast diversity of the analyzed apps, we considered the introduction point, of a smelly test file, as the ratio of the absolute commit position to the total commits of the app:

First Smelly Commit Position = $(\frac{FirstSmellyCommitAbsolutePosition}{TotalAppCommits})$

As shown in Table 6.7, the introduction of a smelly test file occurs earlier on in the project; approximately at the 23% of the apps commits. For each identified first commit of

Table 6.7: Statistical summary on the 1st smelly commit

| **Item** | **Min.** | **1st Qu.** | **Median** | **Mean** | **3rd Qu.** | **Max.** |
|---|---|---|---|---|---|---|
| 1st Smelly Commit Position (percentile) | 0 | 1.5 | 9.1 | 23.6 | 39.7 | 98.3 |
| Smell Types in 1st Commit of a Test File | 0 | 2 | 3 | 2.9 | 4 | 7 |
| Smell Types in 1st Commit of a Smelly Test File | 1 | 2 | 3 | 3.1 | 4 | 7 |

a unit test file, we identified the number of test smells (if any) that were exhibited by these files. As shown in Table 6.7, on average, a unit test file is added to a project with 3 test smells.

Table 6.8: Type of smell occurring in the 1st commit of a smelly test file

| **Smell Type** | **Occurrence in 1st commit** |
|---|---|
| Assertion Roulette | 54.66% |
| Conditional Test Logic | 17.43% |
| Constructor Initialization | 8.78% |
| Default Test | 3.85% |
| Duplicate Assert | 18.47% |
| Eager Test | 37.08% |
| Empty Test | 2.04% |
| Exception Handling | 52.10% |
| General Fixture | 14.67% |
| Ignored Test | 3.66% |
| Lazy Test | 30.64% |
| Magic Number Test | 31.91% |
| Mystery Guest | 7.14% |
| Redundant Assertion | 2.95% |
| Redundant Print | 2.02% |
| Resource Optimism | 3.59% |
| Sensitive Equality | 6.07% |
| Sleepy Test | 1.56% |
| Unknown Test | 25.37% |

To understand the types of smells that occur in a unit test file when the file first starts to exhibit test smells, we analyzed the first smelly version of each unit test file. Our analysis showed that when a test file becomes smelly, on average, three types of smells are added to the test file (Table 6.7). Further analysis showed that *Assertion Roulette* is the frequently occurring smell, followed by the *Exception Handling* smell; both smells occurring in over 50% of the identified smelly files. Table 6.8 lists down the frequency distribution of each smell type occurring in the first smelly commit of the set of unit test files.

Table 6.9: Frequency distribution of smelly commits

| **Item** | **Frequency** |
|---|---|
| ***Position (percentile) of 1st smelly commit*** | |
| 20 | 1.12% |
| 33 | 1.12% |
| 6 | 0.93% |
| 16 | 0.93% |
| 1 | 0.75% |
| ***Smell types in 1st commit of a test file*** | |
| 3 | 23.15% |
| 2 | 20.37% |
| 1 | 15.92% |
| 4 | 14.40% |
| 5 | 8.88% |
| ***Smell types in 1st commit of a smelly test file*** | |
| 3 | 24.49% |
| 2 | 21.70% |
| 1 | 17.21% |
| 4 | 15.26% |
| 5 | 9.38% |

The frequency distribution table, Table 6.9, provide further information on the introduction of test smells. The table provides the top 5 commit positions of a smelly file and the top 5 smell types that initially occur in a unit test file. It was observed that the majority

(approximately 82%) of the unit test files had 1 or more test smell types when the unit test file was added to the project repository.

### 6.1.5 RQ1.6: What is the general trend of test smells exhibited by apps over time?

Next, we investigated the trend of test smells in our studied apps. To achieve this measured how frequently smells increase, decrease or remain at a steady level during the lifetime of an app and for each instance of a smelly unit test file. Additionally, we performed a comparison of the number of unit test files (and the associated test smells) that are updated by developers for multiple apps across a common timeline.

Table 6.10: Statistical summary of smell trend in app and uni test files

| Item | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|---|---|---|---|---|---|---|
| ***Smell trend in apps*** | | | | | | |
| Steady State | 0 | 0 | 2 | 239.1 | 22 | 38650 |
| Smell Increase | 0 | 0 | 0 | 10.76 | 2 | 1451 |
| Smell Decrease | 0 | 0 | 0 | 9.474 | 1 | 1403 |
| ***Smell trend in unit test files*** | | | | | | |
| Steady State | 0 | 0 | 2 | 14.77 | 6 | 1933 |
| Smell Increase | 0 | 0 | 0 | 0.71 | 0 | 292 |
| Smell Decrease | 0 | 0 | 0 | 0.64 | 0 | 291 |

For each unit test file, we obtained the total number of smells that the file exhibited every time it was committed to the repository. We then compared the number of smells exhibited in each version of the file in chronological order, and recorded the number of times the smells in the file increases, decreases or remains the same (i.e., steady). Our finding indicated that the number of smells exhibited by a file remains constant throughout all updates to the file. Next, we calculated the cumulative totals of each type of smell trend for all unit test files of an app. Using this data, we were able to obtain a view of how

frequently smells in an app change over time. As shown in Table 6.10, on average, when a smelly test file undergoes updates during its lifetime, approximately 14 times the smell count remains constant. Similarly, the test smells exhibited by the app as a whole, remains steady during the lifetime of the app.

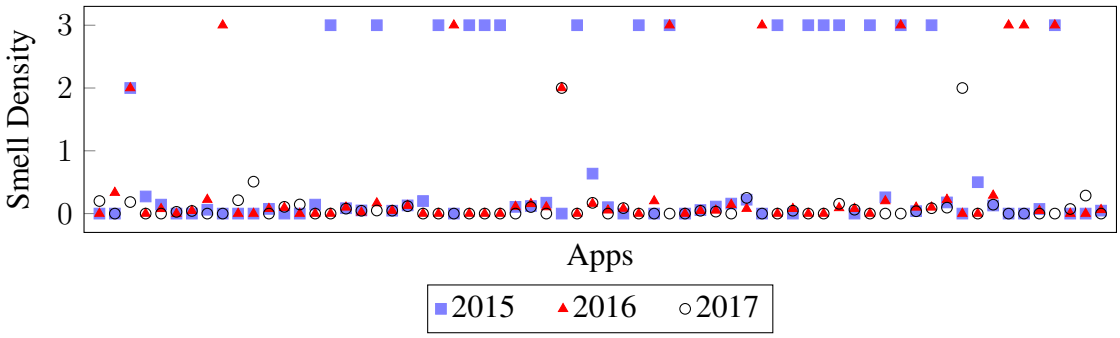


Figure 6.2: Test smell density trend for selected apps across overlapping years

Even though our dataset of Android apps spans across eight years, for this experiment, we selected a date range that contained the same set of apps (i.e., all the commits made for these apps overlap during this time). We identified that the period 2015 to 2017 contained the most number of common apps, 66, over the longest period. Due to the diversity of the apps in our study, we opted to utilize test smell density (i.e., count of total test smells exhibited by test files divided by LOC of the test files) as the comparison metric. For each app, we obtained the test smell density that occurred within each year. In Figure 6.2, apps are represented on the x-axis, while the test smell density values are represented on the y-axis. As time progress, the test smell density, of a majority of the apps, remain more-or-less constant.

### 6.1.6 RQ1.7: Does developer experience have a part to play in the existence of test smells?

Our investigation into the type of users that introduce test smells into unit test files involved determining the experience of the developer at the time of performing a commit of a smelly

file. To this extent, we calculated the Developer Commit Experience (DCE) for each developer for every commit performed by the developer. For each commit made by a developer, we obtain the total number of commits performed by the developer prior to the current commit. We then divide this value by the total number of app commits (regardless of the person who performs the commit). The DEC for developer *a* at commit *i* is calculated as:

$$DCE_{a,i} = \frac{\sum_{n=1}^{i} c_{a,n}}{TotalAppCommits}$$

where $c_{a,n}$ is the $n^{th}$ commit performed by developer $a$.

It is by design that the DCE of a developer changes over the development lifetime of the app. This approach ensures that as the app grows, we can accurately capture the experience of the developer during the apps lifetime. A higher DCE value indicates that the developer performing the commit has contributed more to the development of the app, and hence more experienced.

Table 6.11: Statistical summary of DCE for test files

| **Item** | **Min.** | **1st Qu.** | **Median** | **Mean** | **3rd Qu.** | **Max.** |
|---|---|---|---|---|---|---|
| DCE - Smelly Test File | 0 | 0.01 | 0.04 | 0.08 | 0.11 | 0.41 |
| DCE - Non-Smelly Test File | 0 | 0.02 | 0.06 | 0.10 | 0.18 | 0.49 |

As shown in Table 6.11, there is not much of a difference in average DCE values for developers that commit smelly and non-smelly files. A Spearman rank correlation coefficient test to assess the association between test smells and DCE produced a week negative statistically significant correlation (Table 6.12). Hence, it is our understanding that developers, regardless of project experience, are probably unaware of test smells. They either introduce smells into test files or do not fix smells already in existence in the test files being updated.

Table 6.12: Spearman's correlation calculation on test smells and DCE. Bold indicates that the obtained result was statistically significant (p$<$ 0.05).

| **Variable #1** | **Variable #2** | **Correlation ($\rho$)** |
|---|---|---|
| Total Test Smells | DCE | **-0.06** |

***RQ1 Summary***: Test smells are widespread in the test suites of Android apps with *Assertion Roulette* not only being the most commonly occurring smell, but also having the most number of co-occurrences with other smell types, and also the most common smell type first introduced into a project. We also observed that test smells are introduced early on in the projects lifetime and that the experience of the developer does not influence the presence of smells. Further, when compared to non-Android Java applications, the top four smells occurring in both environments are the same with similar distribution ratios.

## 6.2 RQ2: To what extent does the severity of test smells increase the risk of test files change- and bug-proneness?

To measure the severity of test smells, we define the following hypothesis concerning a unit test files' change-proneness (resp. bug-proneness):

**Hypothesis 1** *The probability of changes (resp. bugs) affecting smelly test files is higher than the non-smelly test files' probability of change (resp. bugginess).*

To verify this hypothesis, we calculated its Odds Ratio (OR), similar to [22], to compare it with the null hypothesis that advocates the higher chances of change (resp. bugginess) of non-smelly test files compared to those of smelly test files. An OR value of 1 means that there is no difference in the probability of change (resp. bug) proneness whether the file is smelly or not, while an $OR > 1$ favorites Hypothesis 1 i.e., the odds of code changes (resp. bugs) to occur within smelly files is higher while an $OR < 1$ rejects Hypothesis 1.

To calculate the ORs, we analyzed our apps repository to extract the needed sets defining the events of each hypothesis. To obtain the dataset for our change-proneness analysis,

we first obtained the number of revisions (i.e., commits) each test file undergoes during the lifetime of the project. Next, for each smell type, we divide the revision resultset into two parts - test files exhibiting the specific test smell and those that do not. To obtain the dataset for our bug-proneness analysis, we collected issues, raised by developers, and labeled with the *bug* tag. GitHub provides developers the ability to close issues using a commit by specifying a predefined keyword in the commit message. The CommitID is then associated with the issue and is available as part of the issues metadata. For each app, we extracted the list of closed issues from the apps issue tracker. Next, we filter in unit test files associated with CommitIDs specified in the issue tracker data. Approximately 23% of the detected unit test files, associated with apps tracking test-related issues, had associated issues. Finally, for each smell type, we clustered the infected files into two groups depending on whether the test file has an issue-related history or not.

Table 6.13 shows the computed OR results corresponding to the bug and change-proneness of smelly test files clustered by smell type. It is interesting to note that ORs and their significance vary depending on the smell type. Test files infected with *Assertion Roulette* are found to be more change-prone and especially bug-prone than test files without smells. This result is not surprising since these infected test files are responsible for testing multiple functionalities and are usually linked to multiple functions. Thus, an issue reported in any of these functions would require going through a large number of assert statements in the test file to fix the related test cases. Also not surprising is the high change and bug probability for test files exhibiting the *General Fixture* smell. This smell originates when the `setUp` method is too general (i.e., test methods do not require all the actions performed by the `setUp` method). As new test methods are added or existing methods modified, to the test class, the `setUp` method will need to be updated to ensure that the test methods execute successfully and not result in unexpected runtime exceptions. Additionally, breaking changes introduced to production methods (possibly due to bug resolution activities), called from the `setUp` method, also impact the OR value of this smell. The *Sleepy Test* smell scored the highest number of change probability; this is because these test files are

Table 6.13: Odds Ratio for bug and change-proneness. Bold indicates that the obtained result was statistically significant ($p < 0.05$).

| Smell Type | Odds Ratio | |
|---|---|---|
| | Change Proneness | Bug Proneness |
| Assertion Roulette | **1.63** | **2.27** |
| Conditional Test Logic | **1.19** | 0.46 |
| Constructor Initialization | **1.40** | 0.28 |
| Default Test | 0.12 | 0.95 |
| Duplicate Assert | **1.36** | 0.72 |
| Eager Test | **1.28** | 0.95 |
| Empty Test | 1.17 | 1.21 |
| Exception Handling | **1.10** | 0.58 |
| General Fixture | **2.19** | **1.09** |
| Ignored Test | 0.75 | 0.45 |
| Lazy Test | **1.47** | 0.83 |
| Magic Number Test | 1.02 | 0.92 |
| Mystery Guest | **1.35** | 0.70 |
| Redundant Print | **1.63** | 1.24 |
| Redundant Assertion | 0.81 | 0.37 |
| Resource Optimism | **2.04** | 0.41 |
| Sensitive Equality | **1.22** | 0.66 |
| Sleepy Test | **2.58** | 0.56 |
| Unknown Test | **1.52** | 0.44 |

using delays to simulate an external event (waiting for a response) prior to continuing with the test execution. This delay is intended by developers to simulate the nature of third party communication gaps and delays (e.g., waiting for server response). This manual and very subjective process is behind the high odds of change associated with this smell. Also, external resources can easily introduce bugs to the test file since developers do not have control over their behavior, which is ironically not tested. Test files exhibiting the *Default Test* smell, on the other hand, has been found to be significantly reluctant to change; a reason for this could be attributed to it being a default test file added by Android Studio, and developers are either unaware of it or do not add app specific test methods to it. Test files

infected with the *Conditional Test Logic* smell are complex by nature as they contain many controls statements (loops, ternary conditional statements, etc.), so they also have shown a higher probability to change over time. However, surprisingly such files are less likely to be prone to bugs. Similarly to code smells' impact, test files infected with *Unknown Test* smell tends to be hard to understand, and thus to maintain.

Table 6.14: Overall Odds Ratio for bug and change-proneness for unit test files. Bold indicates that the obtained result was statistically significant ($p < 0.05$).

| Proneness Type | Odds Ratio |
|---|---|
| Change | **1.28** |
| Bug | 0.57 |

An overall Odds Ratio computation for unit test files (Table 6.14) indicated that test files exhibiting test smells are more sensitive to undergo changes in their lifetime in comparison with their proneness to faults.

***RQ2 Summary***: Our investigation statistically showed that test files exhibiting smells are more likely to change during its lifetime with files exhibiting the *Assertion Roulette* smell highly prone to changes and bugs. However, further research into bug-proneness is required as developers may not always indicate if/when a test file is updated to address a bug.

## 6.3 RQ3: Do test smells act as an indicator of technical debt within the test suite?

As a reminder, technical debt is the implementation of less than optimal solutions to deliver a software product within a less than required timeline or budget. Since there is no consensus on how to measure technical debt, we have decided to extract technical debt instances where developers intentionally admit the existence of a shortage in their implementation, also known as Self-Admitted Technical Debt (SATD). In this exploratory study, we investigate the presence of SATD in unit test files and the degree to which it acts as an indicator of smells in test files. SATD occurs when developers knowingly inject bad programming

practices into the codebase to meet a specific requirement [19]. In this context, we utilized JavaParser to perform an automated (case insensitive regular expression) search of unit test files for the presence of SATD keywords within the files' comments. Our search process utilized the set of SATD keywords defined by [10].

Table 6.15: Distribution of test smell types in unit test files containing SATD

| **Smell Type** | **Distribution Percentage** |
|---|---|
| Assertion Roulette | 15.81% |
| Conditional Test Logic | 8.90% |
| Constructor Initialization | 2.67% |
| Default Test | 0.00% |
| Duplicate Assert | 12.11% |
| Eager Test | 6.29% |
| Empty Test | 0.65% |
| Exception Handling | 10.35% |
| General Fixture | 4.61% |
| Ignored Test | 2.16% |
| Lazy Test | 5.33% |
| Magic Number Test | 10.77% |
| Mystery Guest | 3.20% |
| Print Statement | 0.55% |
| Redundant Assertion | 0.64% |
| Resource Optimism | 2.75% |
| Sensitive Equality | 3.12% |
| Sleepy Test | 0.65% |
| Unknown Test | 9.46% |

Results from our analysis indicated that approximately 10.26% of unit test files contained one or more SATD keywords. These files were distributed across 25% of the analyzed apps. Interestingly, when we analyzed the SATD-based files for test smells, 96% of the files exhibited test smells. From this, we observed that the smell *Assertion Roulette* occurred the most. The high occurrence of this smell is not surprising since it is one of the most frequently occurring test smell in all test files. Table 6.15 provides a complete

Table 6.16: Occurrence of each SATD keyword

| SATD Keyword | Occurrence |
|---|---|
| todo | 77.70% |
| needed? | 10.94% |
| fixme | 6.19% |
| workaround | 2.29% |
| hack | 2.23% |
| unused? | 0.60% |
| wtf? | 0.05% |
| kludge | 0.00% |
| stupidity | 0.00% |
| yuck! | 0.00% |

breakdown on the smell type distribution. We also observed that the SATD keyword 'todo' occurred the most; Table 6.16 lists the breakdown on SATD keyword occurrence. Examples of test cases exhibiting SATD are provided in Listings 6.1, 6.2, 6.3 & 6.4.

```
@Test public void testSetLocale() {
    setUpTypical();
    subject.setLocale(LOCALES[0].getLanguage());
    // todo: how to verify?
}
```

Listing 6.1: Test method exhibiting an Unknown Test smell

```
// FIXME This test doesn't really work as expected
public void testGetLastKnownLocation() throws InterruptedException {
    .....
    // TODO See if that solves sporadic test failures.
    Thread.sleep(500)
    .....
}
```

Listing 6.2: Test method exhibiting a Sleepy Test smell

```
public void testLooksLikeURL() {
    .....
    assertFalse(StringUtils.lastPartLooksLikeURL("abc.def"));
    // TODO: ideally this would not look like a URL, but to keep
    // down the complexity of the code for now True is acceptable.
    assertTrue(StringUtils.lastPartLooksLikeURL("abc./def"));
    // TODO: ideally this would not look like a URL, but to keep
    // down the complexity of the code for now True is acceptable.
    assertTrue(StringUtils.lastPartLooksLikeURL(".abc/def"));
}
```

Listing 6.3: Test method exhibiting an Assertion Roulette smell

```
public void testInvalidMoves() {
    // TODO
    // fail("TODO");
}
```

Listing 6.4: Test method exhibiting an Empty Test smell

Our investigation into SATD also resulted in a brief analysis of refactoring on unit test files. We observed that approximately 3% of all unit test file commits were related to refactoring. We arrived at this conclusion by performing a search on the commit message for the keywords including 'refactor', 'refactoring' and 'refactored'. While we agree that such an approach is not entirely reliable, it does provide a starting point for future investigations. Future studies in this area can include following a similar approach to [39].

***RQ3 Summary***: Our findings indicate that unit test code, similar to production code is also subject to technical debt. Furthermore, the presence of test smells acts as a reliable indicator of technical debt in the test suite. However, further research in this area is needed to understand the precise impact test smells on technical debt. We recommend developers take into consideration these initial findings during development.

## 6.4 RQ4: What is the degree to which tsDetect can correctly detect test smells?

To evaluate the effectiveness of tsDetect in correctly detecting test smells, we performed a qualitative analysis to obtain the precision and recall of our tool. We enlisted 39 subjects from the Department of Software Engineering at Rochester Institute of Technology [1] to manually construct an oracle to which we could compare the detection results of our tool. Subjects included undergraduate and graduate students and faculty members. All the subjects volunteered to help with the experiment and were familiar with Java programming. The experience of these subjects with Java development ranged from 2 to 11 years, which included exposure to developing unit tests. Prior to the commencement of the experiment,

[1] https://www.se.rit.edu/

subjects were provided with a 75-minute lecture on test smells along with reference materials. From our dataset of unit test files, we selected 65 random files by executing the SQLite `Random()` function. Subjects were randomly grouped into groups of 3, to reduce the effect of subjective bias. Each group was provided with ten test files (along with the associated production files). Each group was also provided with a template that they used to indicate the type of test smell that the file exhibited. Duration of this experiment was three days, and all of the 13 groups submitted their results within this period. We performed a cross-validation exercise to reduce the impact of subjects on the evaluation by providing each smell type to at least two groups. Subjects were asked to justify their decisions, and the experiment organizers reviewed their explanations. We next ran our tool on the same set of test files and then compared our results against the oracle. For each smell type, we constructed a confusion matrix and calculated the Precision, Recall, Accuracy, and F-Score. Table 6.17 reports on the correctness of detecting each smell type. For reproducibility purposes, we provide the anonymized qualitative analysis package (files used in the experiment) on our website.

Additionally, without limiting our qualitative analysis of tsDetect to manual validation, we also performed a comparative exercise of tsDetects ability of smell detection against state-of-art detection strategies. Within the list of tools described in Chapter 3, TRex was the only available tool, but its detection rules were specific to identifying violations of the TTCN-3 standards. Among the other non-available tools, we were successful in acquiring Test Smell Detector (TSD), a tool used in a few recent test smell studies [4, 46]. Ideally, the comparison has to be performed with a dataset of manually detected test smells. Since there are no publicly available test smell datasets, we decided to run both tools against a set of manually validated smells, identified in the previous experiment. However, the comparison was limited to smell types recognized by tsDetect and TSD. Our findings indicated that tsDetect is better equipped to detect smells in specific scenarios than TSD. Listed below are some scenarios that we encountered where tsDetect performed better than TSD.

**Assertion Roulette** In Example #1 of Listing 6.5, TSD falsely indicates that the method

Table 6.17: The correctness of tsDetect in detecting test smells

| Smell Type | Precision | Recall | Accuracy | F-Score |
|---|---|---|---|---|
| Assertion Roulette | 95% | 81% | 86% | 87% |
| Conditional Test Logic | 98% | 100% | 99% | 99% |
| Constructor Initialization | 94% | 97% | 94% | 96% |
| Duplicate Assert | 96% | 100% | 97% | 97% |
| Eager Test | 96% | 92% | 93% | 94% |
| Empty Test | 100% | 100% | 100% | 100% |
| Exception Handling | 93% | 97% | 95% | 95% |
| General Fixture | 93% | 92% | 96% | 92% |
| Ignored Test | 97% | 99% | 83% | 98% |
| Lazy Test | 96% | 82% | 89% | 89% |
| Magic Number Test | 100% | 100% | 100% | 100% |
| Mystery Guest | 96% | 100% | 98% | 98% |
| Redundant Assertion | 97% | 100% | 98% | 98% |
| Redundant Print | 89% | 100% | 99% | 94% |
| Resource Optimism | 90% | 100% | 93% | 95% |
| Sensitive Equality | 95% | 100% | 96% | 97% |
| Sleepy Test | 96% | 100% | 98% | 98% |
| Unknown Test | 93% | 100% | 97% | 96% |

exhibits the smell. The detection is not valid as this method only contains one assertion statement. In Example #2, TSD falsely indicates that the test method does not exhibit the smell. The detection is not valid since there are multiple non-documented assertion statements.

```
//Example #1
@Test
public void testEncrypt() throws Exception {
    String xml = readFileAsString(DECRYPTED_DATA_FILE_4_14);
    byte[] encrypted = Cryptographer.encrypt(xml, "test");
    String decrypt = Cryptographer.decrypt(encrypted, "test");
    assertEquals(xml, decrypt);
}

//Example #2:
@Test
public void testTimestamp() {
    Assert.assertEquals("201205292002",ExporterFileNameUtils.getTimeStamp(new Date(1338314522376L), Locale.GERMANY));

    Assert.assertEquals("201205292010", ExporterFileNameUtils.getTimeStamp(new Date(1338315008925L), Locale.GERMANY));
}
```

Listing 6.5: Assertion Roulette Example.

**Eager Test** Example #1 in Listing 6.6, contains multiple calls to the production class, but is not indicated as a smell by TSD. However, it should be noted that the methods are defined in the parent class of the production class. Example #2 is reported as having an Eager Test smell, by TSD since the same production class method 'isArea()' is called twice in the same method. However, it can be argued, that this not constitute to an Eager Test smell since it is the same method and the Eager Test rule definition does not indicate if the called methods have to be different.

```
//Example #1
public void testOnUpgrade() {
    DbHelper helper = createHelperVersion(1);
    assertTablesExist(helper.getReadableDatabase(), TABLES_V1);
    helper.close();

    helper = createHelperVersion(2);
    assertTablesExist(helper.getReadableDatabase(), TABLES_V2);
    helper.close();

    helper = createHelperVersion(3);
    assertTablesExist(helper.getReadableDatabase(), TABLES_V3);
    helper.close();

    helper = createHelperVersion(4);
    assertTablesExist(helper.getReadableDatabase(), TABLES_V4);
    helper.close();
  }

//Example #2:
public void testRelation()
{
    assertFalse(OsmAreas.isArea(new OsmRelation(0,0, null, null)));
    Map<String, String> tags = new HashMap<>();
    tags.put("type","multipolygon");
    assertTrue(OsmAreas.isArea(new OsmRelation(0, 0, null, tags)));
}
```

Listing 6.6: Eager Test Example.

**Lazy Test** The example in Listing 6.7 is not reported as having a Lazy Test smell, by TSD. However, it should be noted that the methods are defined in the parent class of the production class. The existence of the smell in this scenario can be debated.

```
public void testOnCreate_v2() {
    DbHelper helper = createHelperVersion(2);
    assertTablesExist(helper.getReadableDatabase(), TABLES_V2);
    helper.close();
}

public void testOnCreate_v3() {
    DbHelper helper = createHelperVersion(3);
    assertTablesExist(helper.getReadableDatabase(), TABLES_V3);
    helper.close();
}
```

Listing 6.7: Lazy Test Example.

**Mystery Guest** TSD checks for the Mystery Guest smell by performing a text-based

search for ‘File’, ‘db’ and ‘File(’. In the sample code provided in Listing 6.8, there is a string variable that contains the characters ’db’; TSD wrongly identified this test file as containing the smell.

```
@Test
public void testParseRadicalName() {
    String kanjidicStr = " 565F U5ddb B47 S3 V1527 H9 MN8669 MP4.0326 P1-1-2 I0a3.2 Q2233.7 Ychuan1 Wcheon   T2  {
         curving river radical (no.47)}";
        KanjiEntry entry = KanjiEntry.parseKanjidic(kanjidicStr);
    .....
}
```

Listing 6.8: Mystery Guest Example.

***RQ4 Summary***: Through manual verification, we established that tsDetect exhibits a high degree of correctness in detecting smells. Further, we also highlighted instances where tsDetect outperforms an existing test smell detection tool.

# Chapter 7

# Threats to Validity

In this chapter, we present factors that may impact the applicability of our observations in real-life situations. We classify these factors into three categories [61].

**Internal Validity.** We report on the uncontrolled factors that interfere with causes and effects, and may impact the experimental results. The task of associating a unit test file with its production file was an automated process (performed based on filename associations). This process runs the risk of triggering false positives when developers deviate from JUnit guidelines on file naming. However, our manual verification of random associations and the extensiveness of our dataset acts as a means of countering this risk.

Further, the random selection of files/data performed at different stages in the study (either as a means of quality control verification or as support for answering research questions) has a risk of not being representative selections. Even though our lexical-based SATD analysis utilized keywords from a prior published study, it lacked context-sensitivity analysis associated with the keyword. Context-sensitivity will reduce the occurrence of false positives.

**Construct Validity.** Herewith we report on certain challenges that validate whether the findings of our study reflect real-world conditions. We assume change-proneness and bug-proneness as two maintenance measures that reflect a direct maintainability issue with the source code. Still, the variation of code changes frequency can also be triggered by several other factors including adding features or performing API updates. To mitigate this issue, we normalize the change frequency over the project overall lines of code. Also, we collected changes and bugs from various projects to ensure the scalability of our results.

For our qualitative analysis, we distributed the evaluation of each test smell among multiple subjects to avoid subjective bias.

Our detection process can still contain false negatives, which constitutes a threat to our findings, especially given that we aimed to assess the relevance of the newly introduced smell types by measuring their impact on maintenance in general through various empirical experiments. However, our findings have confirmed the usefulness of these introduced smell types. In the future, we will continue to refine the definition of these smells to increase the detection accuracy.

**External Validity.** The detection rules utilized by tsDetect was limited to JUnit based unit tests. tsDetect, at present, does not support other testing libraries/frameworks such as TestNG and Mockito. The analysis was limited to only open-source, Git-based repositories indexed on F-Droid. However, we were still able to analyze 656 apps that were highly diverse in age, category, contributors, size, and rating. Not all the apps that we cloned were available on Google Play. Hence our correlations against Google Play metadata may not be representative.

Similarly, not all apps utilize GitHub to track and manage issues/bugs; and for apps that do use GitHub's issue tracker, not all developers associate a commit to a tracked issue. Furthermore, due to cost/logistical constraints, it was not possible to involve developers of apps, analyzed in this study, to confirm/validate assumptions/findings.

# Chapter 8

# Conclusion & Future Work

The objective of this work is to help developers build and maintain better quality test cases for mobile apps. To do so, we have extended the list of known test smells, based on our experience and other sources. We conducted a set of qualitative experiments to investigate the existence of smells in 656 open-source Android apps, we evaluated the correctness of tsDetect, and we reported the impact of test smells on the bug and change-proneness of test files. Our main findings indicate a substantial existence of test smells in unit test files. Their existence represents a threat to test file's maintainability, as they trigger higher chances of more fix-oriented files updates. Some smell types such as Ignored, Empty and Default Test also serve as an indicator of lack of proper testing discipline in the app. Using tsDetect, we hope developers will be better informed on the type of test smells that occur in Android apps and will be able to avoid introducing them in their test files. We further hope that developers will be encouraged to write unit tests early in the project life-cycle and continue to update the tests as the project evolves.

We plan on extending this study by focusing on expanding the existing set of test smells to include new Android-specific test smells. We also plan on: (1) conducting further analysis of apps to understand the co-occurrence relationship between test smells and traditional code/Android smells, and (2) survey the Android developer community to gauge their understanding of test smells and the importance that they place on fixing test smells over (production) code smells. Additionally, we also plan on extending tsDetect to support other testing libraries/frameworks and also providing a plug-in version of tsDetect for popular IDE's.

# Chapter 9

# Acknowledgement

We would like to thank the authors of the Test Smell Detector for providing us with their tool. We would also like to thank all the participants who volunteered in this project for their assistance and advice.

# Bibliography

[1] Gartner says worldwide sales of smartphones grew 7 percent in the fourth quarter of 2016. https://www.gartner.com/newsroom/id/3609817, Feb 2017. (Accessed on 02/02/2018).

[2] Idc: Smartphone os market share. https://www.idc.com/promo/smartphone-market-share/os, May 2017. (Accessed on 02/02/2018).

[3] Abhijeet Banerjee and Abhik Roychoudhury. Automated re-factoring of android apps to enhance energy-efficiency. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 139–150, New York, NY, USA, 2016. ACM.

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

[5] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012.

[6] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Softw. Engg.*, 20(4):1052–1094, August 2015.

[7] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[8] Manuel Breugelmans and Bart Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *Proceedings of the 1st international workshop on advanced software development tools and Techniques (WASDeTT)*, 2008.

[9] Lisa Crispin and Janet Gregory. *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[10] E. d. S. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, Nov 2017.

[11] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[12] Colton Dennis, Daniel E Krutz, and Mohamed Wiem Mkaouer. P-lint: a permission smell detector for android applications. In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pages 219–220. IEEE, 2017.

[13] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Riccardo Roveda. Towards a prioritization of code debt: A code smell intensity index. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, pages 16–24. IEEE, 2015.

[14] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[15] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (ttcn-3). *Computer Networks*, 42(3):375–403, 2003.

[16] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 322–331. IEEE, 2013.

[17] Michaela Greiler, Andy Zaidman, Arie van Deursen, and Margaret-Anne Storey. Strategies for avoiding text fixture smells during software evolution. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 387–396, Piscataway, NJ, USA, 2013. IEEE Press.

[18] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of android code smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 59–69, New York, NY, USA, 2016. ACM.

[19] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 23(1):418–451, Feb 2018.

[20] Oskar Jarczyk, Błażej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. Github projects. quality analysis of open-source software. In Luca Maria Aiello and Daniel McFarland, editors, *Social Informatics*, pages 80–94, Cham, 2014. Springer International Publishing.

[21] Marouane Kessentini and Ali Ouni. Detecting android smells using multi-objective genetic programming. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 122–132. IEEE Press, 2017.

[22] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory

study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 75–84. IEEE, 2009.

[23] Negar Koochakzadeh and Vahid Garousi. Tecrevis: a tool for test coverage and test redundancy visualization. *Testing–Practice and Research Techniques*, pages 129–136, 2010.

[24] L. Koskela. *Effective Unit Testing: A Guide for Java Developers*. Running Series. Manning, 2013.

[25] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith. A dataset of open-source android applications. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 522–525, May 2015.

[26] D. E. Krutz, N. Munaiah, A. Peruma, and M. Wiem Mkaouer. Who added that permission to my app? an analysis of developer permission changes in open source android apps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 165–169, May 2017.

[27] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[28] Margaret Lewis. *Applied statistics for economists*. Routledge, 2012.

[29] Michael F. Malinowski. *IT Maintenance: Applied Project Management*. Management Concepts, Incorporated, 2007.

[30] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M. Almurshed, Danny Dig, and Carlos Jensen. Understanding code smells in android applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 225–234, New York, NY, USA, 2016. ACM.

[31] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 381–384. IEEE, 2003.

[32] Mika V Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.

[33] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, Sep 2006.

[34] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, Feb 2015.

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

[36] Gerard G. Meszaros. Xunit test patterns and smells: Improving the roi of test code. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 299–300, New York, NY, USA, 2010. ACM.

[37] Mohamed Wiem Mkaouer. Interactive code smells detection: An initial investigation. In *International Symposium on Search Based Software Engineering*, pages 281–287. Springer, 2016.

[38] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide. A robust multi-objective approach for software refactoring under uncertainty. In *International Symposium on Search Based Software Engineering*, pages 168–183. Springer, 2014.

[39] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[40] Helmut Neukirchen and Martin Bisanz. Utilising code smells to detect quality problems in ttcn-3 test suites. *Testing of Software and Communicating Systems*, pages 228–243, 2007.

[41] R. Osherove. *The Art of Unit Testing: With Examples in C#*. Manning, 2013.

[42] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015.

[43] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, Aug 2017.

[44] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 5–14. ACM, 2016.

[45] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, pages 5–14, New York, NY, USA, 2016. ACM.

[46] Fabio Palomba and Andy Zaidman. Does refactoring of test smells induce fixing flaky tests? In *Proceedings of the International Conference on Software Maintenance (ICSME). IEEE*, 2017.

[47] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 33. ACM, 2012.

[48] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.

[49] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley. Scotch: Test-to-code traceability using slicing and conceptual coupling. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 63–72, Sept 2011.

[50] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality.

[51] Romain Robbes. Mining a change-based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 15–, Washington, DC, USA, 2007. IEEE Computer Society.

[52] Chris Smith. The sad state of android: Google removed over 700,000 bad apps last year. http://bgr.com/2018/01/31/google-play-store-malicious-apps/, Jan 2018. (Accessed on 02/02/2018).

[53] Joseph Steinberg. Google removed 700,000 problematic apps from its android play store in 2017. https://www.inc.com/joseph-steinberg/google-removed-700000-problematic-apps-from-its-android-play-store-in-2017.html, Feb 2018. (Accessed on 02/02/2018).

[54] F. Steve and N.P. Steve Freeman. *Growing Object-Oriented Software: Guided by Tests*. Pearson Education (US), 2009.

[55] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and*

*Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

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

[57] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 92–95, 2001.

[58] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.

[59] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 391–400. IEEE, 2006.

[60] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800, 12 2007. Copyright - Copyright IEEE Computer Society Dec 2007; Last updated - 2011-07-20; CODEN - IESEDJ.

[61] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[62] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 306–315. IEEE, 2012.

# Appendix A

# tsDetect - Class Diagram

Depicted in Figure A.1 and A.2 are UML class diagrams of tsDetect.

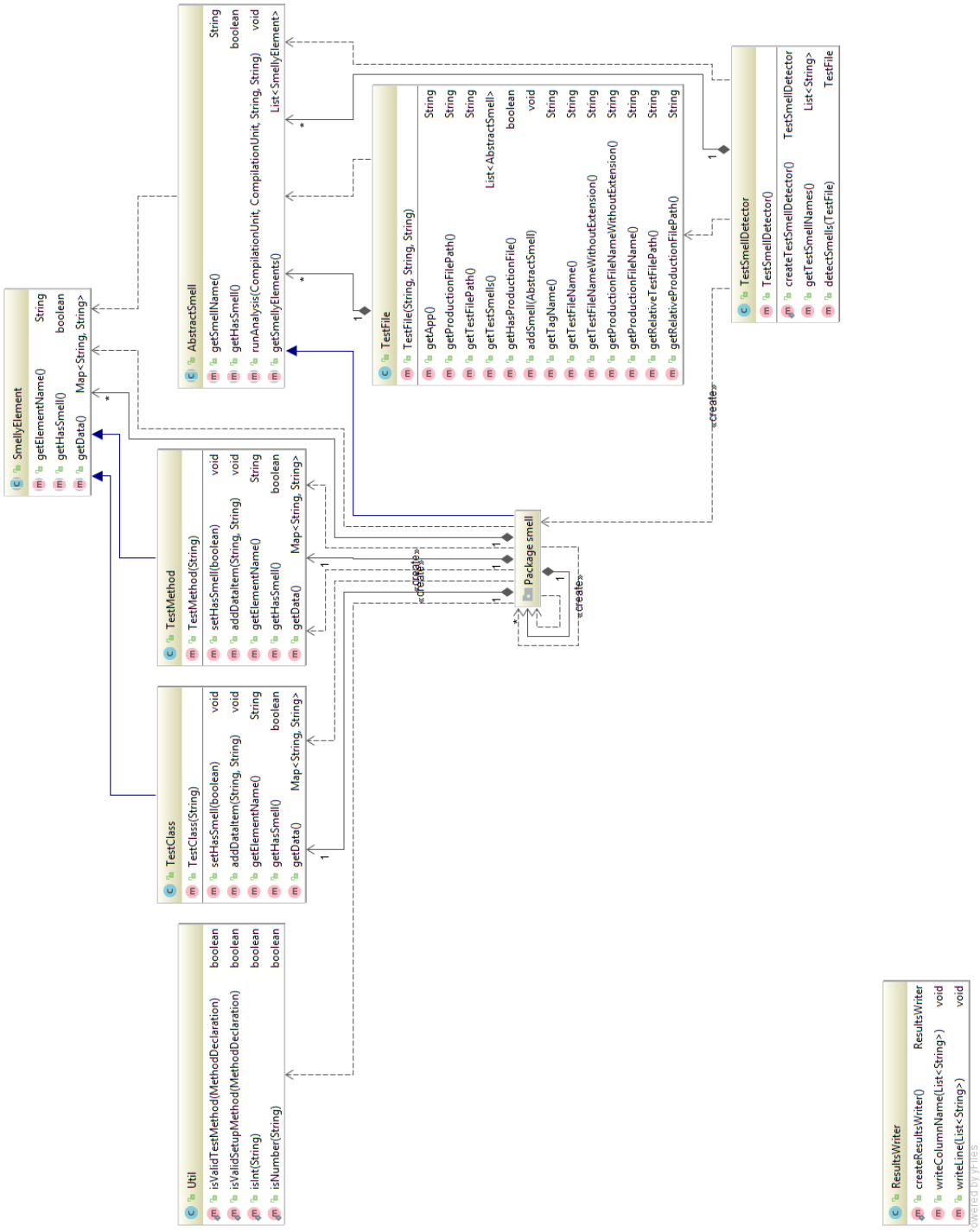


Figure A.1: UML class digram of tsDetect - Overview

Figure A.2: UML class digram of tsDetect - Smell Types