On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study

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

Khalid Almalki
Rochester Institute of Technology
Rochester, NY, USA
ksa8566@rit.edu

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

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

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

Fabio Palomba
University of Zurich
Zurich, Switzerland
palomba@ifi.uzh.ch

ABSTRACT

The impact of bad programming practices, such as code smells, in production code has been the focus of numerous studies in software engineering. Like production code, unit tests are also affected by bad programming practices which can have a negative impact on the quality and maintenance of a software system. While several studies addressed code and test smells in desktop applications, there is little knowledge of test smells in the context of mobile applications. In this study, we extend the existing catalog of test smells by identifying and defining new smells and survey over 40 developers who confirm that our proposed smells are bad programming practices in test suites. Additionally, we perform an empirical study on the occurrences and distribution of the proposed smells on 656 open-source Android applications (apps). Our findings show a widespread occurrence of test smells in apps. We also show that apps tend to exhibit test smells early in their lifetime with different degrees of co-occurrences on different smell types. This empirical study demonstrates that test smells can be used as an indicator for necessary preventive software maintenance for test suites.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software notations and tools*; *Software maintenance tools.*

KEYWORDS

software maintenance, software quality, unit test, test smells

1 INTRODUCTION

Unit test code, just like production source code, is subject to bad programming practices, also known as anti-patterns, defects, and smells. Smells, being symptoms of bad design or implementation decisions/practices, have been proven to be one of the primary reasons for decreasing the quality of software systems; making them harder to understand, more complicated to maintain, and more prone to changes and bugs [1]. In this context, several studies on code smells focus on identifying and detecting what practices, designs, etc. should be considered smells [2, 3] or prioritizing smell correction based on severity with respect to deteriorating the quality of software [4, 5].

The concept of test smells was introduced by van Deursen et al. [6]. Further research in this field has also resulted in the identification of additional test smell types [7], analysis of their evolution and longevity [8, 9], and patterns to eliminate them [10]. However, as described in Section 6, studies around test smells are limited to traditional Java systems. Several studies have designed strategies on how to detect these smells [11–14] or show how the existence of code smells deteriorates the quality of software designs [15, 16], but there are no studies that analyze the existence and distribution of bad testing practices in Android applications (apps). This is in the context of how prolific mobile apps have become to every day life; as of the last quarter of 2018 there were roughly 2.6 million apps available on Google Play [17]. Both users, and developers whose job it is to build and maintain these programs, would benefit from such a study; users will see an improvement in their user experience and developers will have an easier time maintaining apps in the long term. Hence, we have extended the set of existing smells to cover violations of xUnit testing guidelines [18].

To analyze the lifecycle and impact of these smells, we conducted a large-scale, empirical study on test suites utilizing JUnit [19] for 656 open-source Android apps. Further, we defined a series of research questions to support and constrain our investigation to better understand 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 broad goal is to gain a stronger qualitative, and quantitative understanding of test smells in Android apps; to understand how they are similar to and where they diverge from traditional Java systems. In particular, we want to support developers in creating

and maintaining high-quality apps while avoiding increased project costs that are ultimately introduced when developers must manually detect and remove smells. We take steps to 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) validating our proposed smells by open-source developers, and (3) comparing Android apps with traditional Java systems.

Our main findings show that: (1) almost all apps containing unit tests contained test smells introduced during the initial stages of development. The frequency of these smells differs per smell type, while their occurrence is similar to traditional Java systems; (2) test smells, once introduced into an app, tend to remain in the app throughout its lifetime; (3) majority of the developers in our survey agree that our proposed smells are bad testing practices.

2 TEST SMELLS

Test smells are a deviation from how test cases should be organized, implemented, and how they should interact with each other. This deviation indicates potential design problems in the test code [20]. Such issues hurt software maintainability and could also hurt testing performance (e.g., flaky tests [21, 22]). In the subsequent subsections, we provide definitions of our proposed unit test smells and summarize the feedback received from developers on the practicality of the proposed smell types in real-world projects. It should be noted that test smells, like general code smells, are subjective and open to debate [23]. We welcome feedback and extensions to the detection logic for the proposed smells.

2.1 Literature test smells

We now provide a brief introduction to the smells that were part of our study, with more details on each smell available in the work of van Deursen et al. [6]:

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

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

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

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

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

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

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

2.2 Proposed test smells

We extend the existing test smells defined in literature by including a new set of test smells inspired by bad test programming practices mentioned in unit testing based literature [10, 24–26], as well as JUnit, and Android developer documentation [27]. 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 are available on our project website [28].

2.2.1 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. Refer example in Listing 1.

```
/* ** Test method contains multiple control statements ** */
@Test
public void testSpinner() {
    /* ** Control statement #1 ** */
    for (Map.Entry<String, String> entry : sourcesMap.entrySet()) {
        .....
        /* ** Control statement #2 ** */
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            /* ** Control statement #3 ** */
            if (result.testSpinner.runTest) {
                .....
                /* ** Control statement #4 ** */
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.
                            data.get(i));
                .....
}
```

Listing 1: Example - Conditional Test Logic.

2.2.2 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 enable this smell by defining a constructor for the test suite. Listing 2 illustrates an example of this smell.

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

Listing 2: Example - Constructor Initialization.

2.2.3 Default Test. By default Android Studio creates default test classes when a project is created. These template test 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 and violate good testing practices. Problems would also arise when the classes need to be renamed in the future. Example in Listing 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 {
        .....
        assertEquals(beginTime.get(), "200");
    .....
}
```

Listing 3: Example - Default Test.

2.2.4 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 created. As a best practice, 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. Refer to the example in Listing 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: Example - Duplicate Assert.

2.2.5 Empty Test. Occurs when a test method has no executable statements. Such methods are possibly created for debugging purposes without being deleted or contain commented-out test statements. An empty test method 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 altered outcomes as JUnit will report the test as passing. An empty test example is presented in Listing 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 5: Example - Empty Test.

2.2.6 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 custom exception handling code or exception throwing. An example is provided in Listing 6.

```
@Test
public void realCase() {
    .....
    /* ** 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 6: Example - Exception Handling.

2.2.7 Ignored Test. JUnit 4 provides developers with the ability to suppress test methods from running. However, these ignored test methods result in overhead with regards to compilation time and an increase in code complexity and comprehension time. Refer to example in Listing 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 7: Example - Ignored Test.

2.2.8 Magic Number Test. This smell occurs when a test method contains unexplained and undocumented numeric literals as parameters or as values to identifiers. These magic values do not sufficiently indicate the meaning/purpose of the number. Hence, they hinder code understandability. Consequently, they should be replaced with constants or variables, thereby providing a descriptive name for the value. Listing 8 shows an example of this smell.

```
@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 8: Example - Magic Number Test.

2.2.9 Redundant Print. Print statements in unit tests are redundant as unit tests are executed as part of an automated script. Furthermore, they can consume computing resources or increase execution time if the developer calls an intensive/long-running method from within the print method (i.e., as a parameter). Refer example in Listing 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 9: Example - Redundant Print.

2.2.10 Redundant Assertion. This smell occurs when test methods contain assertion statements that are either always true or always false. A test is intended to return a binary outcome of whether the intended result is correct or not, and should not return the same output regardless of the input. Listing 10 highlights an instance of this smell.

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

Listing 10: Example - Redundant Assertion.

2.2.11 Sleepy Test. 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 continue execution. Explicitly causing a thread to sleep can lead to unexpected results as the processing time for a task differs when executed in various environments and configurations. Refer Listing 11 for an example.

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

Listing 11: Example - Sleepy Test.

2.2.12 Unknown Test. An assertion statement describes an expected condition for a test method. By examining the assertion statement, it is possible to understand the purpose of the test. However, it is possible for a test method to be written without 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 a thrown exception when executed. Such programming practice hinders the understandability of the test, as shown in Listing 12.

```
/* ** Test method without an assertion statement and non-descriptive name  ** */
@Test
public void hitGetPOICategoriesApi() throws Exception {
    POICategories poiCategories = apiClient.getPOICategories(16);
    for (POICategory category : poiCategories) {
      System.out.println(category.name() + ": " + category);
    }
}
```

Listing 12: Example - Unknown Test.

2.3 Practicability

To confirm that our list of proposed smells is indicative of problems in the test suite, we surveyed developers that either own or contribute to open source systems that exhibited these smells. From our corpus of systems (described in Section 3), for each proposed smell type, we selected ten random unit test files that were smelly. In total, these 120 unit test files spanned across 100 unique systems.

We conducted the survey over email, and in total, we reached out to 120 developers. We personalized each email to the extent of mentioning the file in the project repository (via the URL of the file hosted on GitHub) and (where applicable) the method name in the file exhibiting the smell. The email also informed participants that the questions were part of an educational (non-profit) research study to understand certain programming practices used by developers when implementing JUnit based unit tests. As participants in this survey were not compensated, we had to ensure that the amount of effort required from each participant was kept to a minimum. To this extent, we limited each participant to a single test file and smell type. Further, we also limited the number of questions to at most three. To prevent bias in the responses, we did not mention negative terms/phrases such as 'smell' and 'bad programming practice' in the email. As each smell type is unique, it was not feasible to have the same question repeated for each smell type. However, within each smell type, the questions were the same. For example, a question on code readability/understandability is appropriate for the smell *Conditional Test Logic* but not for *Redundant Print*. In general, we asked: 1) if the developer could recall as to the reason for implementing a certain construct, 2) if they feel that an alternative (i.e., non-smelly) means of implementation would help improve maintenance and quality, and 3) (where applicable) if certain constructs in their implementation are unnecessary.

We received 50 responses, which approximates to 41.67% of all the developers we contacted. We received responses for each of the proposed smell type except for *Exception Handling*. In most cases, developers confirmed that the programming constructs we highlighted in their code are examples of test smells. It was also interesting to note that some of the developers expressed willingness to rectify the identified smells in their code. However, there were instances where developers would disagree with our findings and prefer to stick to their methodology. As the questions were open-ended, we performed a thematic analysis on the developer responses. In the following subsections, we provide the results from our survey for each smell type. Please refer to our project website for the test files included in our survey.

Conditional Test Logic: While respondents did agree that developers should write unit tests that are easy to comprehend, they could not agree to the outright removal of conditional statements from the test methods. However, they did mention that troubleshooting a failure that occurs within a loop would be problematic if the assertion statement does not provide enough information. They prefer to consider it being smelly or not on a "case by case basis". We also had a scenario where a developer reported that our code snippet was indeed a piece of bad code: "I actually have no idea why that for loop is there. It doesn't do anything but run the test 1000 times, and there's no point in that. I'll remove it."

Constructor Initialization: Interestingly, we observed that the developers who responded to this smell types indicated that they were aware of JUnit fixtures and the common reason for using a constructor over fixtures is "laziness". The respondents also stated that using a constructor is "sloppy" and results in "unexpected behavior". They also unanimously agree that developers should use text fixtures. Once again, based on our finding a respondent made the necessary corrections to the code: " I have already made this change since you pointed it out so the code is clearer now" [29].

Default Test: All respondents agreed that the default tests "serves no concrete purpose" and that it may lead to confusion. We also had a respondent mention that developers should follow a test first approach so that they will be forced to remove the default tests from the onset. Again, we had instances of developers removing these files from their repository due to our survey [30, 31].

Duplicate Assert: We obtained mixed responses to this smell; it comes down to personal preference - some developers preferred to split the assertion statement into separate methods while others did not. However, developers that prefer the latter do mention that they might consider using separate methods depending on the size and complexity of the test. For example, "I might enforce it on some bigger projects." Developers agree that comprehensibility and maintainability are important and believe that adding more

information into the assertion text would help. Looking at the theme in the responses, we observed that having all assertions within a single test method is convenient during implementation (e.g., "It is possible to split that single test into 4 different ones but I'd have to come up with a name for each different case"). They prefer to deal with this smell only when debugging (e.g., "Someone's attention would be drawn to this test case if it failed, and they would look to understand it, potentially including changing it").

Empty Test: Respondents were unanimous that such test methods should be removed from the test suite. However, at the same time, they also feel that such methods can be used to verify the testing framework. Furthermore, a respondent indicated that test coverage is good mitigation for such smells.

Ignored Test: This smell type also had a set of mixed reviews. We had developers mention that they "would not tolerate to have ignored tests in the code" and such tests "should be commented out or removed from a test file". However, some respondents felt ignored tests permit developers to investigate problems or serve as a means for new developers "to understand behavior" and should remain in the codebase. Developers do agree about the overhead in compilation time, but feel that this can be ignored depending on the size of the test suite. Like some of the other smells, this smell too boils down to developer preference.

Magic Number Test: Respondents showed indecisiveness for this smell. While they agreed that the use of constants over magic numbers improve code readability/understandability, they feel that it should not be a blanket rule. For instance, if a numerical value has a "deeper meaning", then a constant should be used so that its "name adds useful information", but not in situations where the meaning of the number is apparent and hence the constant becomes "superfluous". The respondents pointed out specific areas in their test code that could be improved by replacing the numeric values with constants and other instances where such an action is not necessary. In summary, if the meaning of a numeric value can be verified by looking only at the code, then no constant is needed.

Redundant Assertion: A common reason for the existence of this smell is due to developer mistakes, and the respondents did confirm that such code "is not needed", "bad style" and "should probably be removed". We did encounter a respondent who mentioned that as part of their teams test-driven development process, they create a "canary test" [32] "as a sanity test, or for purposes of warming up". These tests can be removed after serving their purpose, but in this instance, the developer opted not to. Interestingly, there were a few respondents that indicated that the code is required for their tests to execute (possibly to support an extreme edge case). Again, we had a respondent report back the instance we highlighted will be marked for removal.

Redundant Print: Not surprisingly, all respondents were unanimous in agreeing that print statements are redundant and do not belong in test suites. Developers primarily utilize such statements for debugging purposes (e.g., "I wanted to check the signature of an object..."), and then forget to remove them. We also had a respondent who confirmed that these statements result in "a waste of resources (cpu+disk space)" and took steps to update their tests to remove such instances [33].

Table 1: Overview of phase-wise data breakdown.

Phase	Item	Value
Collection	Total cloned repositories	2,011
	Cloned apps hosted on GitHub	1,835
	Total commit log entries	1,037,236
	Total 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
Detection	Apps containing test files	656
	Candidate test files detected	206,598
	Test files with a production file	112,514
	Test methods analyzed	1,187,055
	Test smells	
	Test files not exhibiting any test smells	5,915
	Test files with 1 or more smells	175,866
	Test files with only 1 type of smell	22,927
	Test files with 2 to 5 types of smells	95,565
	Test files with 6 to 10 types of smells	33,898
	Test files with over 10 types smells	3,317
	Average number of smell types in a file	3

Sleepy Test: From our set of responses, developers pause execution to simulate/handle animations, load times, and delays between events/activities. It was interesting to note that almost all respondents confirm that there are risks (i.e., inconsistent results) involved with causing a thread to sleep and provided possible instances where their code might fail (e.g., "transition between activities", "machine is slow for whatever reasons" and "run test on real device"). While knowingly admitting the possibility of failure of their test, sometimes developers have no other choice, or the alternative "requires more code". Even setting a high sleep duration is not recommend as it will increase test execution time.

Unknown Test: The majority of the developers are in favor of having assertion statements in a test method, but there are some minor exceptions which fall under edge cases. The respondents confirmed that the missing assertions in their methods were mistakes and blame it on "sloppy coding". We also have an instance where the developer made the necessary correction to the test method based on our survey email [34].

Summary: Our survey has shown that a majority of the developers in our survey confirmed our proposed smells as bad programming practices in unit test files. However, these smells can be subjective and may not apply to all systems. Some of these systems are well established in the community, and correcting these smells might not be feasible. However, we did encounter instances where developers made (or plan to make) the necessary corrections to their test suite based on our findings.

3 METHODOLOGY

For our study, we conducted a two-phased approach that consisted of: (1) data collection and (2) smell detection. In the first phase, we collected project repositories from multiple sources, while in the second phase, we analyzed repositories for the existence of test smells. Table 1 provides an overview of the data collected/analyzed in each phase. The details of each phase are described in the following subsections.

3.1 Data Collection Phase

Similar to prior research [35, 36], for this study, we utilized F-Droid's [37] index of open-source Android apps and 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 or 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, (2) list of all files affected by each commit, and (3) the complete version history of all identified test files and their production files.

3.2 Smell Detection Phase

To detect test smells in our corpus, we implemented an AST-based tool, tsDetect. The tool is open-source and currently supports the detection of the 19 test smells described in Section 2. tsDetect is able to correctly detect test smells with a precision score ranging from 85% to 100% and a recall score from 90% to 100% with an average F-score of 96.5%. More details about tsDetect, including the architecture of the tool along with its test file and smell detection strategy, are available on our project website [28].

4 ANALYSIS & DISCUSSION

4.1 RQ1: How likely are Android apps to contain unit test smells?

We address RQ1 through three sub-RQs, related to various aspects of test smells such as their existence, 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.

RQ1.1: *Are 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. We observed that non-smelly apps contained significantly less unit test files, over 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 since the count of Java files occurring in the lifetime of smelly and non-smelly apps were similar. Hence, a possible explanation for the absence of the test smells in the 21 apps is that these apps had low unit-testing coverage. Table 2 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 in size, so do 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 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 files and test smell occurrence, we observed that the dataset is of a non-normal distribution. Therefore, 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. Not surprisingly, we obtained a strong positive and statistically significant ($p<0.05$) correlation value of 0.90 between the two variables. Therefore, we can reject Null Hypothesis 1 and statistically confirm that test smells exhibited by an app increase as the volume of unit test files in the app increase.

Table 2: Five number summary of the distribution of source code files in apps

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

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

To aid our discussion on the occurrence of test smells, we calculated the distribution of each test smell type from the total quantity of detected test smells (Figure 1); the volume of apps and unit test files that exhibit each smell type (Table 3); and the co-occurrence of test smells (Table 4). 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 in [38], one reason for the high occurrence of the *Assertion Roulette* 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 methods. 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;

Table 3: 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%

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 has a low distribution. We observed that the *Mystery Guest* and *Resource Optimism* 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 exemplary 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.

RQ1.3: *How does the distribution of smell types in Android apps compare against traditional Java applications?*

Prior research on test smells has mostly focused on test smells exhibited by traditional Java applications, and has shown that such systems are not test smell-proof. In this context, we are interested in understanding the degree to which the distribution of the different test smell types differ between Android and traditional Java applications. Similar to our Data Collection Phase, we retrieved a random set of popular (based on stars, subscribers, and forks)

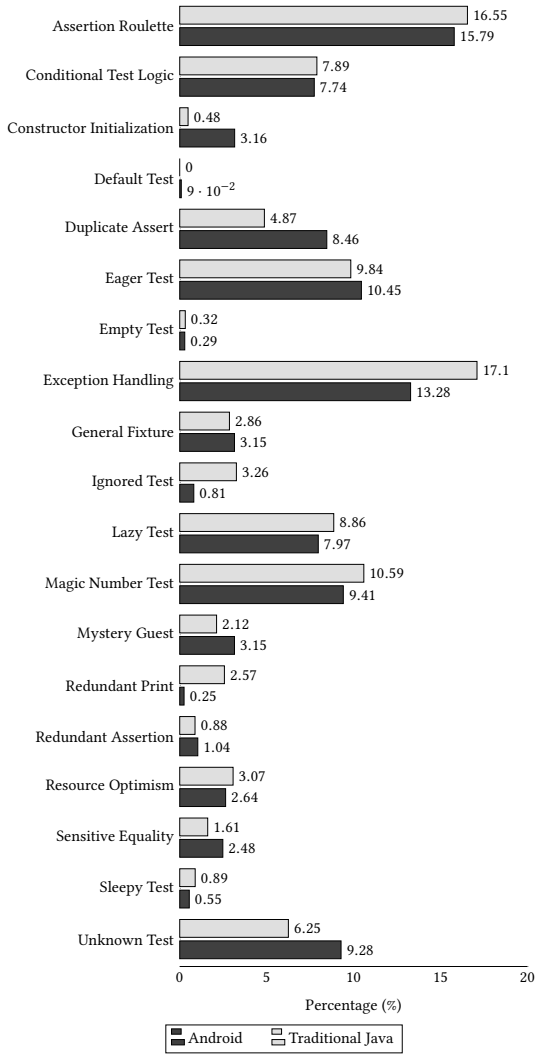

Figure 1: The distribution of the different test smell types in traditional Java and Android applications

traditional Java systems (details available on our website). Next, we ran tsDetect on the version history of all detected unit test files.

Figure 1 shows the distribution of the different smell types in both environments. The graph shows the ratio of occurrence a smell type in an environment when compared against the occurrence of all smell types. Using a percentage instead of actual count values provides a better means of comparison due to size differences of the systems in the experiment. For example, when compared to all smell types, the occurrence of *Assertion Roulette* was 15.79% for Android apps and 16.55% for the traditional Java systems.

It is interesting to note that most smells have a similar distribution in both environments. Further, smells such as *Exception Handling*, *Assertion Roulette*, *Magic Number Test*, *Lazy Test* and *Eager Test* are occurring the most in both environments. This phenomenon is not surprising as these types of smells are not associated with specific API's, but are more of how developers write general

Table 4: 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 |

unit testing code. On the other hand, smells such as *Sleepy Test*, *Mystery Guest* and *Resource Optimism*, are mostly associated with some specific action (such as database/file access or thread manipulation) and hence might only be present when such an action is only required. We did observe a noticeable difference in occurrence for *Constructor Initialization*, *Ignored Test* and *Redundant Print*. The difference in proportion for *Redundant Print* can be attributed to some Android tests being instrumentation based tests and hence developers avoiding the use of print statements. However, further research would be required to explain the differences in the other two smells. 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 [8] we observed that our findings, for the common set of test smells, are also similar.

Summary for RQ1: Test smells are widespread in the test suites of Android apps with *Assertion Roulette* being the most frequently occurring smell and also having the highest number of co-occurrences with other smell types. Furthermore, when compared to traditional Java systems, the top four test smells occurring in both environments are the same and exhibit similar distribution ratios.

4.2 RQ2: What is the general trend of test smells in Android apps over time?

We break down this RQ into two sub-research questions.

RQ2.1: *When are test smells first introduced into the project?*

Our study on the introduction of test smells into a project involves 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.

Table 5: Five number summary on the introduction of 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

For each app in our study, we identified the very first instance of a smelly unit test file was first introduced (i.e., committed) into the apps' project repository. Given the vast diversity of the analyzed apps, we used a ratio based calculation to ensure a standardized means of comparison. In this context, we defined the First Smelly Commit Position (FSCP) as the ratio of the commit position of when the first smelly instance of the file was introduced to the total commits of the app. Formally, we define the First Smelly Commit Position FSCP_f of a file f as follows:

$$\text{FSCP}_f = \frac{C_f}{N} \quad (1)$$

where, C_f is the position in the commit log where the first smelly instance of file f was introduced; and N is the total number of repository commits.

As shown in Table 5, we observed that developers, on average, introduce smelly files earlier on in the project's lifetime– approximately 23% of the way through the total app commits. We also observed that, on average, a unit test file is added to a project with 3 test smell types. Furthermore, when a non-smelly file turns smelly, developers tend to introduce 3 smell types.

An analysis of the smell types occurring in the first smelly instance of a file showed that *Assertion Roulette* is the frequently

Table 6: The type of smell occurring in the 1st commit of 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%

occurring smell, followed by the *Exception Handling*; both smells occurring in over 50% of the identified smelly files. Table 6 shows the frequency distribution of each smell type occurring in the first smelly commit of the test files.

RQ2.2: *How do test smells exhibited by the apps evolve over time?*

To investigate the trend of test smells we 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. For each unit test file, we first, in chronological order, obtained the total number of smells that the file exhibited every time it was committed to the repository. Next, we compared the difference in smell counts between each version of the file, again in chronological order. When calculating the difference, we recorded the number of times the smells in the file increases, decreases or remains the same (i.e., steady). Our findings show 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 7, on average (i.e., 14 times), when a smelly test file undergoes updates during its lifetime, the smell count remains constant. Similarly, the test smells exhibited by the app as a whole, remains steady during the lifetime of the app.

Summary for RQ2: Test smells tend to be introduced early in an Android app's lifetime, and they are likely to remain steady during the lifetime of the app. *Assertion Roulette* and *Exception Handling* are the most common smell types first introduced into a project.

5 POTENTIAL THREATS TO VALIDITY

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. To mitigate this threat, we performed a manual verification of random associations. Further, the extensiveness of our dataset also acts as a means of countering this risk. It also should be noted that 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) may not be representative selections. 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 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 detection accuracy. The detection rules utilized by *tsDetect* were limited to JUnit based unit tests. *tsDetect*, at present, does not support other testing libraries/frameworks. Our study was limited to only open-source, Git-based repositories indexed on F-Droid. However, we were able to analyze 656 apps that were highly diverse in age, category, contributors, and size.

Table 7: Five number summary of the trend of smells in app and unit 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

6 RELATED WORK

Test smells were initially introduced by van Deursen et al., in the form of 11 unique smells [6]. Test smells originate from bad development decisions ranging from the creation of long and hard to maintain test cases to testing multiple production files using the same test case. The same authors found that refactoring test code is different from refactoring production code [6]; demonstrating the value of studying test code apart from production code.

Van Rompaey et al. [39] proposed a set of metrics for the detection of the General Fixtures and Eager Test smells. They aimed to find out the structural deficiencies encapsulated in a test smell. The authors extended their approach to demonstrate that metrics can be useful in automating the detection of test smells [40] and confirmed that test smells are related to test design criteria. Similarly, Reichhart et al. [11] represented test smells using structural metrics in order to construct detection rules by combining metrics with pre-defined thresholds. In other studies, Greiler et al. [13] introduced the General Fixture, Test Maverick, Dead fields, Lack of cohesion of test methods, Obscure in-line setup and Vague header setup smells. Palomba et al. [41] proposed the use of textual analysis for detecting instances of General Fixture, Eager Test, and Lack of Cohesion of Test Methods, showing that it can be more powerful than structural approaches. The impact of test smells has been also shown by researchers. Palomba et al. [21, 22] investigated the impact of test smells on flaky test cases. The experiments confirmed that test flakiness can be caused by test smells in almost 75% of the

cases. Spadini et al. [42] investigated how test smells impact the fault-proneness of production code, showing that classes tested by smelly tests tend to be more fault-prone over their history.

Bavota et al. [15] 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 [43] 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. These results were later confirmed in the context of automatic test case generation [9].

These empirical studies highlight the importance of the community to develop tools to detect test smells and automatically refactor them. Tufano et al. [16] 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. Breugelmans et al. [12] built a tool, TestQ, which allows developers to visually explore test suites and quantify test smells. Similarly, Koochakzadeh et al. [44] built a Java plugin for the visualization of redundant tests. Neukirchen et al. [14] created T-Rex, a tool that detects any violations of test cases to the Testing and Test Control Notation (TTCN-3) [45].

7 CONCLUSION

The objective of this work is to help developers build and maintain better quality test cases for Android apps. To do so, we have extended the list of known test smells and conducted a set of qualitative experiments to investigate the existence of smells in 656 open-source Android apps. Additionally, we validated our proposed smell types with open-source developers. Our main findings indicate a substantial number of test smells in unit test files. Their existence represents a threat to test file's maintainability, as they trigger a higher chance of more fix-oriented file updates. For instance, the existence of *Assertion Roulette* was predominant across test files, and that smell is known to hinder test comprehension. Some smell types such as *Empty*, and *Default Test* also serve as an indicator of lack of proper testing discipline in the app. Finally, we encourage researchers and developers to download and contribute to the extension/improvement of tsDetect [28].

REFERENCES

[1] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 75–84, IEEE, 2009.

[2] M. Kessentini and A. Ouni, "Detecting android smells using multi-objective genetic programming," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pp. 122–132, IEEE Press, 2017.

[3] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[4] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*.

[5] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.

[6] A. Van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pp. 92–95, 2001.

[7] M. Greiler, A. Zaidman, A. v. Deursen, and M.-A. Storey, "Strategies for avoiding text fixture smells during software evolution," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, 2013.

[8] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, 2015.

[9] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. 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*, pp. 5–14, ACM, 2016.

[10] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[11] S. Reichhart, T. Gîrba, and S. Ducasse, "Rule-based assessment of test quality.,"

[12] M. Breugelmans and B. V. 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.

[13] M. Greiler, A. van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*.

[14] H. Neukirchen and M. Bisanz, "Utilising code smells to detect quality problems in ttcn-3 test suites," *Testing of Software and Communicating Systems*, 2007.

[15] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. 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*, 2012.

[16] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. 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, (New York, NY, USA), pp. 4–15, ACM, 2016.

[17] R. Verdecchia, I. Malavolta, and P. Lago, "Guidelines for architecting android apps: A mixed-method empirical study," in *2019 IEEE International Conference on Software Architecture (ICSA)*, pp. 141–150, March 2019.

[18] G. 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, 2010.

[19] JUnit, "A framework to write repeatable tests." https://junit.org/.

[20] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, 2002.

[21] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?," in *Proceedings of the International Conference on Software Maintenance (ICSME). IEEE*, 2017.

[22] F. Palomba and A. Zaidman, "The smell of fear: on the relation between test smells and flaky tests," *Empirical Software Engineering*, pp. 1–40, 2019.

[23] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11.

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

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

[26] F. Steve and N. Steve Freeman, *Growing Object-Oriented Software: Guided by Tests*.

[27] Google, "Android developers." https://developer.android.com/.

[28] S. U. T. Smells. http://testsmells.github.io/, 2018.

[29] https://github.com/mybatis/mybatis-3/commit/88a48ec.

[30] https://github.com/Pawemix/Chronicles/commit/f231d9b.

[31] https://github.com/hussien89aa/QuranOnAndroid/commit/d7c35d6.

[32] P. Hamill, *Unit test frameworks: tools for high-quality software development*.

[33] https://github.com/salvatorenovelli/crawler-service/commit/e11b5eb.

[34] https://github.com/ebean-orm/ebean/commit/b7d5aa3.

[35] D. E. Krutz, N. Munaiah, A. Peruma, and M. W. 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)*, pp. 165–169, May 2017.

[36] 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*, pp. 522–525, May 2015.

[37] F-Droid, "Free and open source android app repository." https://f-droid.org/.

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

[39] B. Van Rompaey, B. Du Bois, and S. Demeyer, "Characterizing the relative significance of a test smell," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pp. 391–400, IEEE, 2006.

[40] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, p. 800, 12 2007.

[41] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 311–322, IEEE, 2018.

[42] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–12, IEEE, 2018.

[43] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Softw. Engg.*, 2015.

[44] N. Koochakzadeh and V. Garousi, "Tecrevis: a tool for test coverage and test redundancy visualization," *Testing–Practice and Research Techniques*, 2010.

[45] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, "An introduction to the testing and test control notation (ttcn-3)," *Computer Networks*, vol. 42, no. 3, pp. 375–403, 2003.