Tracing naming semantics in unit tests of popular Github Android projects

Matej Madeja
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia
matej.madeja@tuke.sk

Jaroslav Porubän
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia
jaroslav.poruban@tuke.sk

Abstract

The tests are so closely linked to the source code that we consider them up-to-date documentation. Developers are aware of recommended naming conventions and other best practices that should be used to write tests. In this paper we focus on how the developers test in practice and what conventions they use. For the analysis 5 very popular Android projects from Github were selected. The results show that 49 % of tests contain full and 76 % of tests contain a partial unit under test (UUT) method name in their name. Further, there was observed that UUT was only rarely tested by multiple test classes and thus in cases when the tester wanted to distinguish the way he or she worked with the tested object. The analysis of this paper shows that the word "test" in the test title is not a reliable metric for identifying the test. Apart from assertions, the developers use statements like `verify`, `try-catch` and `throw exception` to verify the correctness of UUT functionality. At the same time it was found out that the test titles contained keywords which could lead to the identification of UUT, use case of test or data used for test. It was also found out that the words in the test title were very often found in its body and in a smaller amount in UUT body which indicated the use of similar vocabulary in tests and UUT.

2012 ACM Subject Classification Software and its engineering → Semantics; Software and its engineering → Software testing and debugging; Software and its engineering → Software reverse engineering; Software and its engineering → Maintaining software

Keywords and phrases unit tests, android, real testing practices, unit tests, program comprehension

Digital Object Identifier 10.4230/OASIcs.SLATE.2019.1

Acknowledgements This work was supported by project VEGA No. 1/0762/19: Interactive pattern-driven language development.

1 Introduction

Antoniol et al. in [1] argue that a programmer trying to understand an existing system or its code usually goes beyond documentation that is written in a natural language. There are several resources within the documentation, such as document of requirements, user manuals, maintenance book, system manual, etc. These forms of documentation, mostly expressed in a natural language, can facilitate interaction between the system (its source code) and the programmer. The problem with these documents is that each change of source code mostly requires a documentation change as well.

Demeyer et al. [4] indicate that the best reflection of the source code are tests which must remain consistent with the source code during the whole product maintenance. Thanks to that developers can use tests as always up-to-date documentation. Requirements, depending on the development approach, affect the source code and tests in parallel. Test driven development (TDD) [2] drives the development by building tests according to the requirements and the

master (production) code is produced according to tests developed in the first step. In a behavioral driven development (BDD) is the production source code programmed as first and the tests are implemented in addition. Therefore, the requirements are expressed in a particular production or test code.

From the cooperation with companies in our geographical area, during building a testing environment for Android courses [8], we found out that tests are often used to understand the source code by developers engaged in a project. When tests are considered as a source code documentation, the programmer expresses his/her mental model in the source code of the test, and later, another programmer can comprehend the expected functionality from that code. This opens up the possibility to improve the program comprehension through tests.

If a programmer uses tests to understand the application functionality, this process can be very tedious as it is necessary to manually create relations between the test and the production code in programmer's head. If these relations can be formed in the programmer's head, we can assume that creation of these relations could be automated with possibility to enrich the source code, e.g. using comments. Today there are not information about real testing practices used by developers in practice and without this knowledge no one is able to expect how can the test code looks like. There are recommendations, e.g. naming conventions or test smells, whose try to unify the testing process in different projects but the developer can ignore following these conventions. Therefore, the aim of this paper is to found out semantics of vocabulary used in unit tests, which should support program comprehension in the future, especially with focus on open-source Android projects from the GitLab platform. Authors answer the following research questions in this paper:

RQ1: Can a test method or a test class be clearly distinguished by using the word "test" in its name?
RQ2: Does the test title (test method name) contain name of the unit under test (UUT)? Is one UUT tested by multiple test classes?
RQ3: Do words used in the test title clearly describe the status of the application before, during and after the test?
RQ4: Is it possible to evaluate the comprehension of the test body analyzing its statements?
RQ5: Does exist a relation between the test title and test body or UUT body?

In section 2 recommended naming conventions and bad smells with focus on tests are described. Section 3 describes the process of projects selection for analysis, data collection, case study results and possible threads to validity. In section 4 related work is presented, section 5 describes the future research directions and in section 6 results conclusions are pointed out.

2 Best practices of writing tests

To make tests more reliable and effective many authors tried to define practices that help maintain the test code. In the following sections we briefly describe them with focus on our case study.

2.1 Naming conventions

In the listing 1 is an example of a simple test for the `MyUnit` class by recommendations of Beck and Gamma [3]. Authors write about the basic test writing conventions, e.g. for Java programs is expected that one test class tests only one UUT and the test title should

consist of the name of the UUT and the word `Test` at the end (e.g. `MyUnitTest`). This way we can clearly assume which production class will be tested. An integrated development environment (IDE), e.g. *Android Studio*[1], usually automatically offers the test title in the form of described convention, so we can assume that such naming as common.

Listing 1 Example of test class writing practices in JUnit.

```
public class MyUnitTest {
    @Test
    public void testConcatenate() {
        MyUnit myUnit = new MyUnit();
        String result = myUnit.concatenate("one", "two");
        assertEquals("onetwo", result);
    }
}
```

Beck and Gamma also recommend to include the word `test` as first in the test title to distinguish a test method from a tested method. At the same time, the test title should include the name of the UUT from the production code, e.g. test with title `testConcatenate` will test the `concatenate` method. Fixtures may also be used in the tests class and this way we can clearly distinguish fixtures with test method. When generating tests using the *Android Studio* IDE, the names of the generated test methods are identical to the test class methods and without word ”test” in it, but it is still possible to clearly identify the method (e.g. by `@Test` annotation).

However, with the described method of test naming an issue can occur if a developer would like to test one UUT by several tests. Meszaros in his book [9] also mentions testing single UUT by multiple test, where he also describes how to name them, e.g. in the form of use case or test data. An example of such tests are shown in the listing 2.

Listing 2 Use case or data information the in test method name.

```
public void testAddWithoutExtraParam(){
    int res = (new MyUnit()).add(2, 8);
    assertThat(res).isEqualTo(10);
}

public void testAddWithDouble(){
    int res = (new MyUnit()).add(2.88, 7.12, true);
    assertThat(res).isEqualTo(10);
}
```

2.2 Test smells

The concept of smells in the source code was introduced by Fowler et al. [5] and it refers to the bad design and implementation of the source code which subsequently exacerbates its sustainability. Van Deursen et al. [13] on the basis of the findings of Fowler et al., define the smells for tests. For example, tests should not be dependent on external services, external data sources, should be unambiguous, etc. In this article authors focus on one of the smells called *Assert Roulette*, i.e. one test should contain only one `assert` method call. Otherwise, if an unsuccessful assertion in the test occurs it is difficult identify the error and it also aggravates the comprehension. In the listing 3 can be seen an example of mentioned test smell where `assert` statement is used multiple times and if the test fails by an assert, it will be difficult to identify the cause.

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

Listing 3 Assert Roulette test smell.

```
public void testFlightMileage() {
    Flight newFlight = new Flight(validFlightNumber);

    // verify constructed object
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);

    // setup mileage and assert
    newFlight.setMileage(1122);
    assertEquals(expectedKilometres, newFlight.getMileageAsKm());
}
```

3 Case study

Based on the recommendations in the section 2 we decided to find out how developers write tests in practice, focusing exclusively on unit tests in popular open-source Android projects. In the following sections realized case study is described.

3.1 Method

The decision to use open-source Android projects for the case study was done especially for ability to compare results with user interface (UI) tests in the future. Authors think that UI tests can suitably complement the source code with the real use cases that are normally done by end users. However, in this case study analysis of UI tests is not important because we would like to focus on the context of the test title with the name of UUT and the bodies of both (production and test methods). UI tests are mostly associated with the source code only by UI elements identifiers, so analysis of an UI test meaning will need to be evaluated in relation with the entire production class or multiple classes which are called during the test execution. According to *test pyramid* [10] concept, we can expect unit, integration and UI tests in Android projects.

3.1.1 Projects selection

According to Pham et al. [11] we can expect presence of tests in popular projects. That's the reason for selection the most popular Android projects on Github. Open-source projects take advantage of developing software in a transparent environment which motivates developers to produce quality source code because their code is often reviewed by project owner or other developers. This is the reason why developers in a transparent environment use coding conventions more often.

We selected projects from 3 different blogs (DZone[2], Aritra's Musings[3], MyBridge[4]) devoted to the Android open-source popular projects. By merging all 3 sources we received a total of 52 projects. Projects were ranked by popularity using Github API[5], i.e. by the number of project stars. Because the relations between the test and the UUT have been done manually the projects sample was limited to the 5 most popular projects.

[2] https://dzone.com/articles/amazing-open-source-android-apps-written-in-java
[3] https: // blog. aritraroy.in/20-awesome-open-source-android-apps-to-boost-your-development-skills-b62832cffafa4
[4] https://medium.mybridge.co/38-amazing-android-open -source-apps-java-1a62b7034c40
[5] https://developer.github.com/v3/

3.1.2 Data collection

Firstly a manual semantic analysis has been carried out. For each project we tried to comprehend the test, identify the UUT and create a relation between them. We analyzed 617 tests in total. Detailed statistics can be seen in the table 1.

Table 1 General stats of manually analyzed data for particular project.

Position	Project	Production classes	Production methods	Test classes	Test methods
1	Shadowsocks client	6	7	6	8
2	iosched	16	40	16	77
3	Android-CleanArchitecture	17	22	17	29
4	Plaid	37	71	39	180
5	ExoPlayer	49	98	53	323
	SUM	**125**	**238**	**131**	**617**

The relations between the tests and a particular UUT were created using a simple web application in a relational database to have related units (UUT and test) prepared for keyword comparison between the test title and other source code (method body of the test and the UUT). Analysis of a test included the following steps:

1. Comprehension of the test and production code functionality to exactly identify the UUT.
2. Saving the test method body, its name and the class name in which it is located.
3. Saving the UUT method body, its name and the class name in which it is located.
4. Creation of the found relation between the UUT and the test.

The entire data collection lasted 120 hours. Initially it was planned to use the Rompaey's and Demeyer's approach [12] which claims that UUT identification is possible by catching the last method call before the `assert`. However, this solution has proved to be ambiguous because multiple assert calls in a single test suggested several target classes. Another problem was the fact that the last called method was not always the right UUT. In listing 4 there is a specific example from our data collection where the `toString` method is not the target of test but the `fromReader` is the right one. That is the reason why relations between collected data have been done manually.

Listing 4 The last called method is not always the target method for a test.

```
expected = Acl().fromReader(INPUT1.reader()).toString();
assertEquals(expected, "testing string");
```

Some of the tests tested several UUT at once as a result of assert roulette test smell. In this case multiple relations with the production code have been created. If the test contained multiple calls of the `assert` method, the number of calls have been recorded, too, to evaluate the average occurrence of assertions in the test. Also the assertions placed within a help method have been included. If the assert was in the loop the number of its execution has been calculated depending on the test data.

This case study would like to find out if the words included in the test title occur in the test or UUT body. It is considered only the first level of the code (means only main bodies of UUT and test have been saved). E.g. if the UUT named `myMethod` calls in its body an external method `externalMethod`, the body of the external method `externalMethod` is not included for test title words analysis. An exception was a case if the UUT body contained a

call to a single foreign or parent method (such as in the listing 5, the `constuctor` method with three parameters would be used). Following rules have been used for saving the method body for the analysis:

- from the production code the method body with the highest semantic value for the tested functionality,
- from the test code body of the particular test method.

The highest semantic value was determined by the the observer comparison of UUT and test body, i.e. which UUT accurately implements tested functionality.

Listing 5 For keyword analysis is used method with real method statements.

```
public function constuctor(int param1, int param2){
    this(param1, param2, true);
}

public function constuctor(int param1, int param2, boolean param3){
    // real method statements
}
```

3.2 Results

RQ1 is focused on the ability to clearly identify the test or test class. All test classes from analysis included "test" word in the class name, so we could always clearly identify a test class by its name. The word ”test” in the test method title was found in 384 of 617 tests so in most cases is possible to use the method name to determine test. The name of the test method is also influenced by JUnit framework which annotates tests using `@Test` annotation so it is possible to identify a unit test using the annotation. However, the use of the word ”test” may be an appropriate habit for other languages in which the code can not be annotated.

At the same time, there was observed the relation between the word ”test” and the full name of the UUT in the test title. If word ”test” is placed at the beginning of the test title then it expresses meaning: ”test *of something*”. E.g. in the test called `testReadMethod` a man would be able to predict that it is a *test of read method*. Even 124 tests included the UUT name immediately after word ”test” so this metric can partially help with UUT identification.

▷ Claim 1. It is possible to exactly identify test class in Android projects by searching for ”test” word in the test class name. Unit tests written in JUnit are annotated by `@Test` annotation, nevertheless the general naming convention of using word ”test” in test title was found in 62% of tests.

To answer the **RQ2** it was tried to look for the occurrence of words of UUT method name in the test title. As can be seen in the table 2, in the 303 tests (49%) the full name of the UUT was found in the test title, so it was possible to precisely identify the UUT. If the name of the test is composed of multiple UUT names using the last called method before `assert` [12] is possible to identify the UUT more precisely. This way is possible to help the developer to comprehend the meaning and the purpose of the test.

Looking for the partial UUT name in the test title we can get at least a partial link between them. For example, for the UUT with name `onRemoveClicked` was the test title `testRemoveReservation`. As can be seen, the use case of the test is to remove the reservation. If the method is in the `Reservation` class than we can exactly create the relation between the

Table 2 The occurrence of the UUT method name in the test method name.

Name of prod method name included in test method name									
fully included		partially included*							
yes	no	0 w	1 w	2 w	3 w	4 w	5 w	6 w	7 w
303	314	168	191	170	87	10	11	1	2
SUM		472							

*) w = word/words;

test and the UUT. The name of each UUT was divided by the Java *camel case* conventions (e.g. `camelCaseMethod` produces 3 words: camel, case, method). Since the use of *snake case* convention has been observed in some tests (such as `snake_case_method`) this convention was also accepted. In the table 2 we can see that in 472 tests the partial UUT name was found in the test title, which makes 76% of the analyzed tests.

Developers test a class by 2 or more test cases very rarely. Only 3% of all UUT were tested by 2 different test classes. This occurred only in the *ExoPlayer* project where the tested object was in the first class tested directly and in the second class with impact of the various callbacks. For example, if there was a test class `TsExtractorTest` and `TsExtractorSeekTest`, their meaning was as follows:

- `TsExtractorTest` - tests of the basic functionality of the object.
- `TsExtractorSeekTest` - test of the object during expected changes using callbacks.

▷ Claim 2. Unambiguous UUT identification based on the test title can be performed on approximately half of the tests in Android projects and in about 76% of tests is possible to predict the UUT at least partially. An UUT is typically tested by one test class, rarely it is tested by multiple test classes for easier understanding of tested object manipulation.

To answer the **RQ3** manual grouping of the frequently occurring words in test names has been carried out. Words used in test title can affect the reader's comprehension. The table 3 contains a complete list of all the keywords found that can help with automation of test code meaning. By the nature the developer tries to name the test in the way which helps him or her quickly identify the cause of test failure in the future. When a keyword is found in the test name it is possible to assume the content of the data contained in the test title and use them to produce complete sentences in natural language based on predefined templates.

During the analysis it has been noticed that observer comprehended the test easier when the "_" character was used in the title. This character has been used as a delimiter of the semantics, e.g. between the data used in the test and the expected test result. If the delimiter was not part of the test then the readability of the method name was worse and many times there was no clear intention in the test title. For example, the test named `testPeekFullyAtEndThenReadEndOfInput` may have two meanings:

- indicates the entire use case that is a prerequisite to perform the test
- or the test verifies whether during `PeekFullyAtEnd` call the end of input read occurs.

From the test body it is possible to determine the expected behavior but using the delimiter in the test title makes the meaning straightforward. The meaning of some keywords affects the use of other keywords in front of it. As can be seen in the table 3 the most common occurrence was for `when` and `with` keywords. It means that it is possible to read the test prerequisites and data used for testing from the test title quite often.

▷ Claim 3. Developers in test titles often describe the use case of the test, the testing data or the expected result. Occurrence of keywords `test`, `testShould`, `when`, `after`, `if`, `then`, `with`, `is`, `has` and `return/returns` in the test can enable to detect the UUT state before, during and after the test execution, as well as the data used for the test.

Answering the **RQ4** should evaluate the quality of the tests and find out whether the control flow statements are used in the body of test. In the table 4 we can see that 4 different test evaluation approaches were used in the tests. The quality of a test case can be defined as the ability to detect a source code defect. Developer should be able to comprehend the cause of the failure from assertion message. Most often the `assert` statement was used but in 200 tests this command was used multiple times. In case of test failure the error identification is unclear (assert roulette test smell). We counted assertions with and without loops for every test separately. If there was used a loop in the test we calculated the number of assertions according to loop iterations counted from the amount of testing data. We had one big data test that executed `assert` statement 3,794,020 times. In general the following statements are used for test evaluation:

1. `assert` – classical approach,
2. `verify` – to verify calling the correct method on the object after an action,
3. `try-catch` – to verify whether right exception has been thrown,
4. `exception throw` – if an exception is thrown, test failed.

In the case of a multiple exception throw occurrence in a test it would not be considered as test smell because exceptions could be uniquely identified by an exception message.

According to the best practices for testing [9] developers should avoid loops and consequently other control flow statements of particular programming language because these statements create an unstable and obscure test full of smells. In the table 5 we can see the usage of these statements in tests which negatively affect the simplicity of test comprehension and the test failure identification. As we can see, the average assertion count with loops is 6,155, most likely due to the above mentioned big data test, but 30 tests contained more than 10 assertions in the body in total.

▷ Claim 4. Developers often (in our study 46%) create tests that contain multiple verification of UUT which worsen exact and quick identification of UUT. This state directly affects the use of control flow statements in the test body where occurrence of the assert roulette test smell is very high.

In order to find out how much information we can find from the test method name in the UUT or test body (**RQ5**) we created a 20% distribution of words included in test title covered by UUT or test body (see figure 1). Even 94% of the tests contain more than 20% of the test title words in the test body and 57% covers more than 60% of words. The coverage of words from the test title in the UUT body was a little bit worse but not negligible. In 64% of all tests titles was the coverage of corresponding UUT bodies greater than 20% indicating that the test title meaning use similar vocabulary as test which can help to establish the relationship between the source code and the test.

▷ Claim 5. The words included in test title are used mainly in the test body and partly in the corresponding UUT body. Based on the similar vocabulary it is possible to combine the semantic of method calls in the tests and the UUT.

Table 3 Observed keywords which can be used to uniquely identify the purpose of the test from its name.

Keyword	Test method example*	Semantic	Occurrence
test	**test***Read*	what is tested or particular action on object (use case)	370
testShould	**testShould***FailWhenExecute*WithNullObserver	expected test result or behaviour	2
when	**when***NoIntervalsTriggered*_thenMapFn_isOnlyCalledOnce	description of test prerequisite (data or use case)	82
after	testSeekBack**After***ReadingAhead*	description of test prerequisite (data or use case)	15
if	testDownloadLicenseFails**If***NullInitData*	description of test prerequisite (data or use case)	9
then	whenNoIntervalsTriggered_**then***MapFn*_isOnlyCalledOnce	expected result after condition met (if occurred immediately after "when")	3
	testStartsIn0Minutes_**then***HasNullTimeUntilStart*	expected result after condition met (if occurred immediately after "test" and "_" occurred before)**	3
	testPeekFullyAtEnd**Then***ReadEndOfInput*	consequence of method calls in test (if occurred immidiately after "test")	2
with	comment_**with***NoComments*	description of used data for test	85
is	whenNoIntervalsTriggered_thenMapFn_**is***OnlyCalledOnce*	expected result or object state	21
has	testSkipInAlternatingTestSignal_**has***CorrectOutputAndSkippedFrameCounts*	expected result or object state (at the begining of name or exactly after "_")	3
return/returns	testReadDuration_**returns***CorrectDuration*	expected result or object state	40

* Bold text shows the keyword, italic text object to which the keyword refers.
** "_" $\Longrightarrow$ semantic delimiter between UUT, input data, use case etc., used in 322 tests of analyzed data.

Table 4 Statements used for test result evaluation.

	Statement/evaluation of test failure				
	assert (with loops)	**assert** (without loops)	**verify**	**try-catch**	**exception throw**
Tests using particular statement	542	542	77	42	13
Tests using multiple same statements	249	249	26	4	0
Average per test	6155.79	2.16	0.194	0.076	0.021

Table 5 Java language control flow statements used in test methods' bodies.

	Statement				
	for	**foreach** (enhanced for)	**while**	**if**	**switch**
Tests using particular statement	16	7	9	8	0

3.3 Threads to validity

In the performed case study 5 very popular Android projects have been analyzed whose do not include all projects in the world. Therefore, it is possible that the results and claims may not be accurate and adaptable for other projects. With focus on the Android platform the results may be affected by the naming conventions of the selected platform which considerably influence the naming of the methods in the production code or the final implementation. At the same time, the case study was focused on open-source projects that may have a different nature and quality compared to proprietary projects that we did not have access to. The source of the projects was the Github platform. Other collaborative tools can have a different impact on the motivation to write quality code which could also affect the results.

The number of tests between projects was uneven as it is not possible to guarantee the same number of tests in each project. In the future, it would be advisable to limit the maximum number of tests per project so the coding style of a particular team code did not affect the results in the undesirable way.

Because creation of relations between tests and the production code have been done manually some faults could occurred while comprehending the source code and during creation of relations between UUT and test. For more accurate relations between the analyzed data it would be necessary to increase the number of observers in the study.

Analysis of the production code and tests was performed only at the first level of the method call, i.e. only main bodies of the tests' methods. Content and semantics of helper methods and test fixtures have not been included in the analysis, so in case of RQ5 can be meaning of the code hidden in other levels of method calls, too.

4 Related work

Kochhar et al. [7] conducted a case study on open-source Android projects in which authors looked at understanding of testing culture in selected projects. They used F-droid[6] as source of open-source Android projects and they analyzed 600 projects in total. Authors conclude

[6] https://f-droid.org/

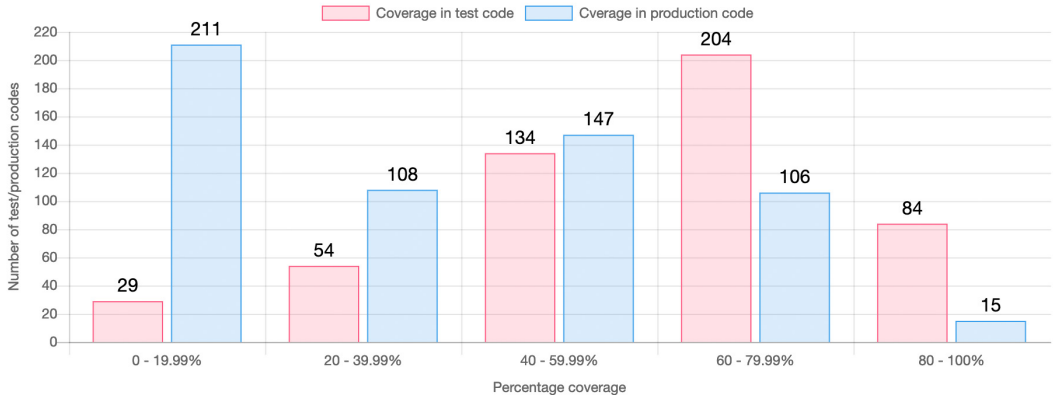

Figure 1 Coverage of words from test method name in body of test or production code.

that many Android projects are poorly tested. They found out that only about 14% of the apps contained test cases and only about 9% of the apps had executable test cases with coverage above 40%. At the same time, they describe what testing tools developer use in practice, however, developers often prefer to test their apps manually. In this paper we analyze the general practices of writing unit tests in open-source Android projects, regardless of the testing framework.

Gellenbeck and Cook [6] devoted their work to the influence of naming variables and procedures on program comprehension. They claims that beacons, sets of key program features, help developers to comprehend an unfamiliar program. The impact of procedure naming on comprehension has been explored on the production code of binary search and sorting procedures. Authors found out that both meaningful procedure and variable names are severest as beacons to high-level comprehension. In this paper we look at naming methods as well, but with respect to the UUT and the test body.

In [12] Rompuy and Demeyer describe the existing strategies for searching the UUT from the test code. However, we have found out that these strategies could not be used in our case study. Described strategies in [12] relatively inaccurately determine the UUT because they depend on the particular structure of the test. By combining these strategies it could be possible to identify the UUT more precisely which is a challenge for future research in this area. The results from our case study complement their strategy based on naming conventions with the real practices of writing tests which can improve the UUT identification.

Pham et al. [11] observed the behavior of developers and their motivation of making tests in a transparent GitHub environment. They contacted 16,000 project developers from whom they received 569 filled out questionnaires of Github members. They found several strategies that software developers and managers can use to positively influence test behavior in their projects. Authors report on the challenges and risks caused by this and suggests guidelines for the promotion of a sustainable test culture in software development projects.

5 Future Work

In the future we would like to compare the results from unit tests with the semantic of UI tests. Since UI tests have less connection to the production source code and they execute functionality from a user perspective it is possible to assume that they will contain the particular user stories that are expected during the real application use in production. However, for analyzing UI tests we will have to track the calls during the program runtime

and in the real context of the device, so creating connections to the source code could be more complicated.

We plan to create a more general overview of real testing practices in different programming languages to maximize the generalization of results. We would like to identify the differences in testing for different languages to find propose features for new IDE tools supporting program comprehension which can have a great impact on the efficiency of creating and maintaining the source code. Our results show that the tests contain many smells that have a negative impact on the comprehension of the program. Based on these results, it is possible to focus on warn the programmer about a test smell occurrence and navigating him/her in better naming of test methods whose name can greatly improve the comprehension.

On the other hand, there exist other techniques that can be used to find semantic relations between UUT and test. In the future it can also be analyzed acronyms, partial words or synonyms to reach more accurate semantics. It is also appropriate to consider approaches that improve program comprehension using code refactoring, e.g. renaming methods names and other identifiers.

With exact determination the UUT from the test it will be possible to enrich the source code with additional information obtained from tests, e.g. how many times the method is tested and in which use cases it is used. If a developer changes the production code then it is possible to notify the programmer that it is necessary to update a specific test that could have been affected.

6 Conclusion

This paper presents a case study on real practices of writing tests in open-source Android projects. For the case study 5 very popular projects from the Github platform have been selected and by manual data collection the relations between the test and the UUT were created. We analyzed 617 tests in 131 test classes and 238 production methods in 125 classes.

There was examined the consistency of the words used in the title of the test and the target UUT. The word was identified based on the Java camel case naming conventions. It was found out that 76% of the tests contain at least the partial UUT name in the test title. At the same time, we tried to identify whether the UUT is tested by only one test class and if not why the developer creates multiple test classes for the same UUT. From the study we claim that developers test an UUT with different test classes rarely and mostly use multiple test classes to distinguish work with the object in the test.

According to the best practices the word ”test” should be found in the test name for its clear identification. It was found out that only 61% of the tests included word ”test” in its name. JUnit is mostly used for unit testing in Java projects which use `@Test` annotation to denote test. This annotation practice influenced results of performed case study in way of the ”test” word occurrence in the test title. Developers in addition to the `assert` statement also use `verify`, `try-catch` and `throw exception` to evaluate the success or failure of the test. In the tests a high incidence of assert roulette test smell has been found which negatively influences the clear identification of UUT. Control flow statements in the test bodies make test difficult to comprehend and complicates creation of relations to the production code.

When creating tests developers use keywords and patterns which can help us to identify the UUT or use case of the test. This work created a summary of all observed keywords used by developers. Importance of individual keywords may depend on their position against other keywords. There was also observed that despite the Java camel case writing convention testers use the ”_” character to separate semantically related data groups in the test title.

In the test title words coverage analysis in the UUT or test body we found out that 57% of the tests had a coverage of more than 60% which means that a short description of the test functionality can be often found in the test title, i.e. particular use case. When comparing the words from the test title to the UUT, the coverage is smaller, but 64% of UUT bodies covers more than 20% words of test title. From the above is possible to claim that the body of the test and the body of the UUT method uses the similar vocabulary.

Despite the fact that this case study focused solely on open-source Android Github projects, the results obtained in this paper can help to develop new and more reliable methods of identifying UUT from the test. Achieving more accurate UUT identification can help to enrich the production code with information from the tests in the future and it has potential to improve the program comprehension.

References

1. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, Oct 2002. doi:10.1109/TSE.2002.1041053.
2. Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
3. Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.
4. Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Elsevier, 2002.
5. M. Fowler, K. Beck, J.C. Shanklin, E. Gamma, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999.
6. Edward M. Gellenbeck and Curtis R. Cook. An investigation of procedure and variable names as beacons during program comprehension. Technical report, Corvallis, OR, USA, 1991.
7. P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015. doi:10.1109/ICST.2015.7102609.
8. Matej Madeja and Jaroslav Porubän. Automated testing environment and assessment of assignments for android mooc. *Open Computer Science*, 8(1):80–92, 2018.
9. Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.
10. Godfrey Nolan. *Agile Android*. Apress, 2015.
11. R. Pham, L. Singer, O. Liskin, F. F. Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 112–121, May 2013. doi:10.1109/ICSE.2013.6606557.
12. B. V. Rompaey and S. Demeyer. Establishing traceability links between unit test cases and units under test. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 209–218, March 2009. doi:10.1109/CSMR.2009.39.
13. 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.