12-2021

Test Naming Failures. An Exploratory Study of Bad Naming Practices in Test Code

Zachariah Wigent
zxw4320@rit.edu

Test Naming Failures. An Exploratory Study of Bad Naming Practices in Test Code

by

Zachariah Wigent

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

Supervised By

Dr. Mohamed Wiem Mkaouer

Dr. Christian Newman

Department of Software Engineering

B. Thomas Golisano College of Computing and Information Sciences

Rochester Institute of Technology

Rochester, NY

December 2021

Committee Approval:

_____

Dr. J Scott Hawker Date
SE Graduate Program Director

_____

Dr. Christian Donald Newman Date
Assistant Professor

_____

Dr. Mohamed Wiem Mkaouer Date
Assistant Professor

To my family and friends, for their continued support and encouragement.

Acknowledgments

Accomplishing this work would not have been possible single-handedly. I am grateful to the many individuals who supported, advised, and encouraged me throughout this process.

My thanks to my advisors, Dr. Mohamed Wiem Mkaouer and Dr. Christian D. Newman, for their dedication and guidance with this research. This work would not have been possible without their knowledge and experience.

Thank you to all the friends and colleagues who provided me with encouragement and support throughout my Maser's program. Many thanks also to my family for their continued support and encouragement in pursuing my Maser of Science in Software Engineering.

Abstract

Unit tests are a key component during the software development process, helping ensure that a developer's code is functioning as expected. Developers interact with unit tests when trying to understand, maintain, and when updating code. Good test names are essential for making these various processes easier, which is important considering the substantial costs and effort of software maintenance.

Despite this, it has been found that the quality of test code is often lacking, specifically when it comes to test names. When a test fails, its name is often the first thing developers will see when trying to fix the failure, therefore it is important that names are of high quality in order to help with the debugging process.

The objective of this work was to find anti-patterns having to do with test method names that may have a negative impact on developer comprehension. In order to do this, a grounded theory study was conducted on 12 open-source Java and C# GitHub projects. From this dataset, many patterns were discovered to be common throughout the test code. Some of these patterns fit the necessary criteria of anti-patterns that would probably hinder developer comprehension. With the avoidance of these anti-patterns it is believed that developers will be able to write better test names that can help speed the time to debug errors as test names will be more comprehensive.

Contents

List of Figures

List of Tables

Introduction

Unit tests are a key component during the software development process, helping ensure that a developer's code is functioning as expected. Developers interact with unit tests when trying to understand, maintain, or update code. Good test names are essential for making these processes easier to carrier out consistently and quickly, which is important considering the substantial costs and effort of software maintenance [20].

Despite this, it has been found that the quality of test code is often lacking [24]. One of these areas that is often lacking is test names. When a test fails, or when the test base requires maintenance, the test names are the first thing developers will generally attempt to understand before they apply changes to the test or the code being tested. If test names are poor quality, developers will need to spend time reading the code and determining how the test's actual behavior is related to its name [36].

Determining whether a test name is high- or low-quality is difficult due to the subjective nature of identifier names. However, there are some objective ways to measure the quality of a name. Specifically, when an identifier name is contrary to the behavior of the entity it represents. That is, if a test name is contrary to the test behavior, we can objectively say that the name can be improved. Thus, this thesis aims to discover patterns in test names that may limit developer comprehension using grounded theory to help us gain a qualitative and quantitative perspective on what a normal test name looks like and what test name anti-patterns look like. These anti-patterns are common practices used by developers that may be detrimental to their comprehension

of test code. The goal of this thesis is to create a taxonomy of test naming patterns and anti-patterns based on an empirical, grounded-theory study.

Paper Structure Section 2 provides an overview of the related work. Section 3 discusses the research objectives. Section 4 goes through the methodology used to conduct this study. The results are discussed in section 5. The validity aspects of the study are discussed in section 6 and the paper is concluded in section 7 with a summary of the results and a discussion of future work.

Related Work

Several studies have looked at how to improve software maintenance through either detecting bad programming practices [2, 4, 8, 25, 30, 31, 33], or correcting them [5, 6, 7, 9, 10, 10, 11, 12, 13, 29, 34].

This section discusses related work, concerning (i) the importance of method names (Section 2.1), (ii) patterns in method names (Section 2.2), (iii) patterns in test methods (Section 2.3, and (iv) grounded theory (Section 2.4).

2.1 Importance of Method Names

Identifiers are an important aspect of programming comprehension, and they are often the starting point for program comprehension as shown by Merlo et al. [26], Caprile and Tonella [18,19], and Anquetil and Lethbridge [14]. Further, Wu and Clause [36] show that test method identifiers are often the first thing a developer will look at when trying to understand a test failure. Lin et al. [24] looked at the quality of identifiers in test suites and compared them to production identifiers, showing that identifiers in test suites are of poor quality. Automatically generated test suites demonstrated even more quality concerns.

2.2 Patterns in Method Names

Linguistic anti-patterns were defined by Arnaoudova et al. [16] as patterns in code identifiers and comments that are misleading to the developer. Arnaodova et al. evaluated these anti-patterns in another study [15] where it

was found that the majority of developers perceived these patterns as poor practices that should be avoided. Further research has been done on how often these linguistic anti-patterns occur [17], how to detect them [1] and how they can be used to improve automated naming tools [3].

In an empirical study on 5,000 open-source projects, Zhang et al. [39] observed that nouns, verbs, and adjectives are three of the most common part of speech tags utilized by developers in crafting identifier names. Newman et al. [28] found similar results looking at grammar patterns in identifier names developers used to describe program behavior. They observed a set of grammar patterns included: noun phrases as one of the most common grammar patterns, function identifiers are more likely to be represented by a verb phrase, and collection types frequently utilize a plural head-noun. Peruma et al. [32] looked at grammar patterns in test method names and found that certain words are frequently used with specific code behaviors. Høst and Østvold [23] proposed a set of naming rules based on their examination of unusual method names. These rules take utilize part of speech tags along with the return type, control flow, and parameters of the method to detect naming violations based on their set of rules.

2.3 Test Method Patterns

Zhang et al. [37] used natural language techniques to parse test method names in order to automatically generate templates for the test methods. They used the action phrase and the predicate phrase found in the test method name in order to generate their templates with over 80% accuracy. In another

study [38] they took the body of a test method and generated descriptive names for that test method. In their approach, the authors analyzed the statements within the test method to determine the action, expected outcome, and scenario under test. An approach to automatically generate short descriptive test method names was developed by Daka et al. [20] based on API-level coverage goals. The results of using these naming patterns were evaluated by surveying 47 students and found to be as descriptive as manually created test names. Wu and Clause [36] utilized a set of test patterns to identify non-descriptive test method names and provide developers with information for a more descriptive name. These test patterns allow for the extraction of the action, predicate, and scenario from the current test name and body so they can be evaluated for descriptiveness.

2.4 Grounded Theory

Grounded theory is a research method that allows for systematic and evidence based development of theories. This research pattern was developed by Glaser and Strauss [21] in order to create theories rather than validate existing ones. Applying this to software engineering is becoming more and more common but has many challenges as noted by Klaas-Jan et al. [35]. They analyzed 98 computer science articles that claimed to use grounded theory and found them to be lacking, leading them to develop guidelines for future researchers using grounded theory in software engineering. Socio-technical grounded theory is a specific branch of grounded theory developed by Hoda [22] in order to provide concrete guidelines for research in software engineering.

Research Objective

3.1 Motivation

Test code and its comprehension is an important part of the software development cycle. Test names are often the first clue that developers have to assist them in determining what happened during a test failure [36] or what parts of the system have already been tested when they are attempting to add or maintain tests, so high-quality identifier names are highly important to developer productivity and comprehension. There have been many studies on what qualifies as a good test name [20, 24, 24], but no studies that have explicitly explored whether test names have their own linguistic anti-patterns. An increased understanding of test linguistic anti-patterns can highlight how developers use test names to understand behavior. It can also highlight the difference between production code and test code anti-patterns, making it very important to fully document and understand test code anti-patterns so that we may then compare how test and production code differ or are similar in how their names convey code behavior.

3.2 Contribution

The main contribution of this study is the creation of a comprehensive taxonomy of unit test naming patterns. These patterns have then been analyzed in order to find common patterns, and anti-patterns, based on available open source test suites. This in-depth analysis of the test naming patterns to find

the common flaws in test naming conventions is a valuable addition to test naming research.

3.3 Research Questions

These patterns in unit test names were analyzed through a grounded theory study in order to answer the following research questions:

- **RQ1: What test naming trends can be identified by examining the name of a test and comparing it to its implementation?** This question seeks to find if unit test names that can be categorized into specific patterns. In order to find anti-patterns in test naming, it first needs to be shown that test names follow specific patterns. This study found a variety of trends that commonly occur in test code.

- **RQ2: Are there test naming patterns that violate the found trends?** The study found that there are test naming trends that are in opposition to one another. There are also many common patterns that are used, but not followed in many cases.

Methodology

Grounded theory is a research method that allows for systematic and evidence-based development of theories. Socio-technical grounded theory [22] is a specific branch of grounded theory that was used to conduct this study on naming patterns within test code. The first step in this process is to conduct a small literature review and come up with initial research questions that will evolve as the study progresses. After this review data collection begins along with the coding and memoing of the data. Once enough data is collected analysis is done to see what conclusions can be drawn and the research questions are updated accordingly. The dataset for this study included 12 projects, shown in figure 4.2, with a total of 457 tests analyzed.

4.1 Literature Review

A light literature review was conducted in order to ensure that the topic would be a viable research contribution. The review was kept small in order to prevent existing concepts from influencing the patterns that are discovered during the study. This allowed for an area to be chosen and research to begin so that more literature could be analyzed as the theory emerged to help validate the results found. The review is divided into three areas: the importance of method names, grammar patterns found in method names, and patterns in test method names

The review showed that while many studies have been conducted on the good practices and patterns used in test naming, there have been no studies

that have specifically looked at anti-patterns present in test names. Anti-patterns have been discovered in other areas of source code [16], but this research is lacking when it comes to test code.

4.1.1 Importance of Method Names

Identifiers are an important aspect of programming comprehension, and they are often the starting point for program comprehension as shown by Merlo et al. [26], Caprile and Tonella [18,19], and Anquetil and Lethbridge [14]. Further, Wu and Clause [36] show that test method identifiers are often the first thing a developer will look at when trying to understand a test failure.

4.1.2 Patterns in Method Names

In an empirical study on 5,000 open-source projects, Zhang et al. [39] observed that nouns, verbs, and adjectives are three of the most common part of speech tags utilized by developers in crafting identifier names. Newman et al. [28] found similar results looking at grammar patterns in identifier names developers used to describe program behavior. They observed a set of grammar patterns included: noun phrases as one of the most common grammar patterns, function identifiers are more likely to be represented by a verb phrase, and collection types frequently utilize a plural head-noun. Høst and Østvold [23] proposed a set of naming rules based on their examination of unusual method names. These rules take utilize part of speech tags along with the return type, control flow, and parameters of the method to detect naming violations based on their set of rules.

4.1.3 Test Method Patterns

Zhang et al. [37] used natural language techniques to parse test method names in order to automatically generate templates for the test methods. They used the action phrase and the predicate phrase found in the test method name in order to generate their templates with over 80% accuracy. In another study [38] they took the body of a test method and generated descriptive names for that test method. In their approach, the authors analyzed the statements within the test method to determine the action, expected outcome, and scenario under test. Wu and Clause [36] utilized a set of test patterns to identify non-descriptive test method names and provide developers with information for a more descriptive name. These test patterns allow for the extraction of the action, predicate, and scenario from the current test name and body so they can be evaluated for descriptiveness.

4.2 Data Collection & Analysis

There are two main aspects to data collection in a socio-technical grounded theory study: coding and memoing. Coding is the process of taking raw data and capturing it in a way that best captures its essence and meaning. Memoing is the documenting of the researcher's thoughts and ideas regarding the emerging concepts, categorizing them, and looking for links between them [22].

The first step in this research was to do open coding on random test code to see what data and patterns may arise. After this was done, a template, seen in figure 4.1, that better captured the main factors that impacted test naming

# of Test in Population	16035
# of Tests Coded	457
# of Memos Created	317
# of Patterns Discovered	11
# of Anti-Patterns Discovered	4

Table 4.1: Overview of the data collected

was constructed to begin more advanced coding. This template is designed to gather all relevant project details, so the test method can be properly traced to its source, collect all necessary test naming information, and collect relevant information about the contents of the tests. The aspects of the template were continuously improved in order for the template to accurately capture whether there was an anti-pattern or not. This stage of the data collection took a total of 9 weeks with approximately 50 tests coded per week and a resulted in a statistical sample with a 95% confidence level with a 4.52% confidence interval.

Projects were selected at random from previous research into curating repositories for research [27] along with popular GitHub repositories. These

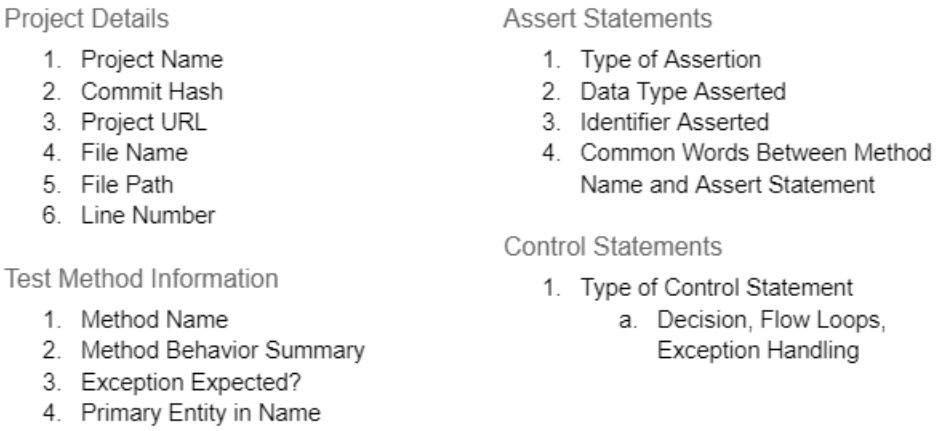

Figure 4.1: The template followed when analyzing each test method.

Project Name	Contributors	Language	KLOC
Bazel	726	Java	1900
druid	150	Java	585.1
Dubbo	363	Java	359.6
ExoPlayer	200	Java	2900
HBase	360	Java	1400
jclouds	230	Java	739.6
Keycloak	507	Java	2900
NewPipe	616	Java	141.9
Pulsar	460	Java	2500
FileSystem	30	C#	14.7
MonoGame	314	C#	252.4
VIPR	8	C#	39.9

Table 4.2: Projects surveyed for test data

projects are listed in table 4.2. For each project the template was applied to an average of 37 tests across multiple files in order to gather an appropriate dataset.

Once these more advanced codings were collected it was possible to begin memoing the data. The first step was just to document for each coding what the factors influencing the method name were, if any. These were then placed into a spreadsheet and compared to one another in order to find potential patterns. This analysis was done manually by searching through and finding memos that documented similar phenomenon with test method names. These groupings of basic memos were then further analyzed and refined in order to come up with actual patterns that were common among many of the test names was done. The resultant pattern set from this analysis is described in section 5.1.1. Once these initial patterns were documented, they were compared and further analyzed to see if they represented good coding practices or what the

researcher would consider anti-patterns. The anti-patterns that resulted from this analysis are described in section 5.2.1.

Analysis & Discussion

The goal of this research was to find common anti-patterns that occur in test names. In order to accomplish this, a study was conducted to find general test naming trends, which were used to help identify potential test naming anti-patterns. The rest of this section goes over the results of the data analysis and the research questions.

5.1 RQ1: What Test Naming Trends can be Identified by Examining the Name of a Test and Comparing it to its Implementation?

There were many naming trends discovered throughout the course of this study. These trends showed themselves throughout many of the projects and test files analyzed. The rest of this section describes the various trends that were discovered as a result of this study.

5.1.1 Discovered Test Naming Patterns

"Test" Prefixed on the Method Name

One common pattern that was seen in 54% of the tests analyzed was the method name starting with "test". This shows a common practice of many developers when coming up with tests name is to explicitly state that a method is a test.

"And" in Test Name

There were 13 tests found with the word "and" in their method name. The majority of these tests were larger functions that contained multiple asserts showing that having "and" in a test name leads to a specific test behavior. An example of this is shown in figure 5.1 which shows the *relativeURIsAndContexts* test method which has multiple asserts on the *populate* method.

```
@Test
public void relativeURIsAndContexts() throws Exception {
    PathBasedKeycloakConfigResolver resolver = new PathBasedKeycloakConfigResolver();

    assertNotNull(populate(resolver, "test")
            .resolve(new MockRequest("http://localhost/test/a/b/c?d=e", "/a/b/c")));

    assertNotNull(populate(resolver, "test")
            .resolve(new MockRequest("http://localhost/test/a/b/c?d=e", "/a/b")));

    // means default context and actually we use first segment
    assertNotNull(populate(resolver, "test")
            .resolve(new MockRequest("http://localhost/test/a/b/c?d=e", "/test/a/b/c")));

    assertNotNull(populate(resolver, "test/a")
            .resolve(new MockRequest("http://localhost/test/a/b/c?d=e", "/b/c")));

    assertNotNull(populate(resolver, "")
            .resolve(new MockRequest("http://localhost/", "/")));
}
```

Figure 5.1: Example test method with "and" in the test name.

"All" in Test Name

"All" was not a commonly found word in test names, but it was strongly tied to a specific test behavior. While only seen in 3 of the tests analyzed, each of the tests either had loops or iterated over a collection in some way. This showed that "all" is a word used by developers when a test has some sort of loop or iteration on data. Figure 5.2 shows the *testAll* function from the jclouds project, which loops over the *apiMetadata* object.

```
@Test
public void testAll() {
   Iterable<ApiMetadata> apisMetadata = Apis.all();

   for (ApiMetadata apiMetadata : apisMetadata) {
      if (apiMetadata.getName().equals(testBlobstoreApi.getName())) {
         assertEquals(testBlobstoreApi, apiMetadata);
      } else if (apiMetadata.getName().equals(testComputeApi.getName())) {
         assertEquals(testComputeApi, apiMetadata);
      } else {
         assertEquals(testYetAnotherComputeApi, apiMetadata);
      }
   }
}
```

Figure 5.2: Example test method with "all" in the test name.

"Exception" in Test Name

The word "exception" was seen in 9 test names, and each of these tests were expected to throw or test for a specific exception. This is a logical pattern to find, as it makes clear the test behavior is to test for an exception. *BuildMani-*

fest_ThrowsInvalidOperationException_WhenTryingToAddAFileWithTheSame NameAsAFolder is a method in the FileSystem project shown in figure 5.3 which exhibits this behavior.

"Throws" in Test Name

Very similar to having "exception" in the test name, "throws" in the test name was present in 7 tests that threw exceptions. These patterns often occurred together but did occasionally occur separately, which is why they are counted as two separate patterns. *BuildManifest_ThrowsInvalidOperation Exception_WhenTryingToAddAFileWithTheSameNameAsAFolder* is a method in the FileSystem project shown in figure 5.3 which exhibits this behavior.

```
[Fact]
public void BuildManifest_ThrowsInvalidOperationException_WhenTryingToAddAFileWithTheSameNameAsAFolder()
{
    // Arrange
    var task = new TestGenerateEmbeddedResourcesManifest();
    var embeddedFiles = CreateEmbeddedResource(
        CreateMetadata(Path.Combine("A", "b", "c.txt")),
        CreateMetadata(Path.Combine("A", "b")));

    var manifestFiles = task.CreateEmbeddedItems(embeddedFiles);

    // Act & Assert
    Assert.Throws<InvalidOperationException>(() => task.BuildManifest(manifestFiles));
}
```

Figure 5.3: Example test method with both "exception" and "throws" in the test name.

Test Names Contain the Name of the Function Under Test

Having the name of the function under test included in the test method name is a common practice among the tests analyzed. This practice allows the test runner to know what function is being tested by any given test, which helps with comprehension. An example from the ExoPlayer project is shown in figure 5.4 where the method *open* is being tested in the *requestOpen* method.

```
@Test
public void requestOpen() throws HttpDataSourceException {
  mockResponseStartSuccess();
  assertThat(dataSourceUnderTest.open(testDataSpec)).isEqualTo(TEST_CONTENT_LENGTH);
  verify(mockTransferListener)
      .onTransferStart(dataSourceUnderTest, testDataSpec, /* isNetwork= */ true);
}
```

Figure 5.4: Example test method where the object under test is in the test name.

Test Names State the Expected Test Result

Many of the tests that were analyzed stated the expected results of the test. This practice ensures that the person running a test knows the result that is expected on a test failure. Figure 5.5 shows an example from the Bazel project where an empty string is the result tested for in the function *emptyStringYieldsEmptyList*.

Test Names State the Required Inputs

Stating the inputs for a test is a pattern that should help to improve developer comprehension. If a developer knows the inputs or preconditions for a

```
@Test
public void emptyStringYieldsEmptyList() throws Exception {
  assertThat(converter.convert("")).isEmpty();
}
```

Figure 5.5: Example test method with the expected result stated in the test name.

test method, it makes the method behavior much more clear. For example, the method *testIsValidDirectoryPathWithEmptyString* from the NewPipe project, shown in figure 5.6, clearly states that the expected input is an empty string.

```
@Test
public void testIsValidDirectoryPathWithEmptyString() {
    assertFalse(FilePathUtils.isValidDirectoryPath(""));
}
```

Figure 5.6: Example test method with inputs stated in the test name.

Single Word Test Name or "Test" and a Single Word Test Name

A pattern that was seen among 40 test names was for them to contain either a single word or a single word and the word "test". This pattern was shown to have a variety of different test behaviors, ranging from very simple to highly complex. An example of this is shown in figure 5.7. The *testHas* function just states the method under test, *has*, but the body of the method tests multiple different types of inputs.

```
@Test
public void testHas() {
  Assert.assertTrue(put.has(FAMILY_01, QUALIFIER_01, TS, VALUE_01));
  // Bad TS
  Assert.assertFalse(put.has(FAMILY_01, QUALIFIER_01, TS + 1, VALUE_01));
  // Bad Value
  Assert.assertFalse(put.has(FAMILY_01, QUALIFIER_01, TS, QUALIFIER_01));
  // Bad Family
  Assert.assertFalse(put.has(QUALIFIER_01, QUALIFIER_01, TS, VALUE_01));
  // Bad Qual
  Assert.assertFalse(put.has(FAMILY_01, FAMILY_01, TS, VALUE_01));
}
```

Figure 5.7: Example test method where the test name is a single word.

Object Under Test is Specified in the File Name

There were some test files, specifically those in the Vipr project where this pattern occurred for all tests, that had good descriptions, but the object under test was unclear. These tests were often structured in a *it returns X* pattern where *it* is the file name. An example of this is shown in figure 5.8. The test method *It_returns_an_odcm_model* is clear about the return type, but the object that should be making this return is ambiguous. With the file name *Given_a_valid_edmx_when_passed_to_the_ODataReader* it becomes clear that the object under test is the *ODataReader*.

Enumerated Test Names have Similar, But Different Functionality

There were at least 10 tests found that had enumerated names. These are tests with similar names, but only differing by an enumeration number or letter. All of these tests tended to have similar functionality within the same

```
[Fact]
public void It_returns_an_odcm_model()
{
    var testCase = new EdmxTestCase();

    var odcmModel = _odcmReader.GenerateOdcmModel(testCase.ServiceMetadata());

    odcmModel
        .Should()
        .NotBeNull("because a valid edmx should yield a valid model");
}
```

Figure 5.8: Example test method with the object under test specified in the file name.

file, but they tested slightly different things. The example shown in figure 5.9 shows one such instance of this.

5.2 RQ2: Are There Test Naming Patterns that Violate the Found Trends?

Of the many patterns discovered, some are considered to be anti-patterns. These are patterns that the researcher feels limit developer comprehension or are in opposition to some other patterns found. These anti-patterns are discussed in the rest of this section.

```
@Test
void testMethodA() {
    if (invocationCountA++ < 1) {
        throw new IllegalStateException("Sample failure to trigger retry.");
    }
}

@Test
void testMethodB() {
    if (invocationCountB++ < 1) {
        throw new IllegalStateException("Sample failure to trigger retry.");
    }
}

@Test
void testMethodC() {
    if (invocationCountC++ < 1) {
        throw new IllegalStateException("Sample failure to trigger retry.");
    }
}
```

Figure 5.9: Example test method with an enumerated test name.

5.2.1 Potential Anti-Patterns

"And" in Test Name

Having "and" in the test name is considered an anti-pattern as these methods tend to be large functions testing multiple different things. A true unit test should only be testing one type of functionality, but these "and" functions are often testing many test cases.

Single Word Test Name or "Test" and a Single Word Test Name

Single word test names are not descriptive, and test names need to be descriptive in order to provide comprehensive value to the developer. A single word does not offer enough information about what a test is doing, which forces a developer to look at the test in order to understand it. The range of behavior that was found with tests that exhibited this pattern is another reason this is considered an anti-pattern. Methods following this pattern were shown to be very short, containing just a single assert statement, or very long, being well over 50 lines. This showed that there was no real pattern between the behavior inside the test and the test name, other than the potential for the one word to be the method under test.

Object Under Test and is Specified in the File Name

This could be considered a valid pattern. Many of the methods following this pattern were very comprehensive as long as the file name was known. However, it is considered an anti-pattern because it contradicts two of the other found patterns: test names contain the name of the function under test and test name states the required inputs. While both the object under test and sometimes potential preconditions were found in the file name for test following this pattern, the *it returns X* pattern is still unclear to someone who is unaware of this test naming scheme.

Enumerated Test Names have Similar, But Different Functionality

Enumerated test names are a known bad practice. It is almost impossible to know the difference in what they are testing without looking at their implementation. This defeats the main purpose of a test name in helping a developer comprehend a test. Even with some more descriptive test names for the example in figure 5.9, like *testSampleFailureA* and *testSampleFailureB*, it would not be possible to know the difference between these methods without looking at their implementations.

5.3 Takeaways

Tests Names Share Common Words With Their Implementation

Out of the 12 projects and 457 tests surveyed, it was shown that 52% of the assert statements analyzed had words in common with the method name. This shows that the patterns *test names contain the name of the function under test*, *test names state the expected result*, and *test names state the required inputs* are all fairly common. These patterns are also viewed as some of the best practices to follow for improved developer comprehension.

Test Names Should be More than One Word

Single word test names are not descriptive, and test names must be descriptive to provide comprehensive value to developers. A single word does not offer enough information about what a test is doing, which forces a developer to look at the test in order to understand it. About 9% of the tests analyzed

fell into the *single word test name or "test" and a single word test name*, which show that this pattern is not followed in the majority of test names.

Some Words Correlate to Specific Types of Behavior

While many of the grammar patterns shown did not have a high rate of occurrence, when they did occur it was almost always linked to a specific type of behavior. This was true for all of the following patterns: *"and" in test name*, *"all" in test name*, *"exception" in test name*, *"throws" in test name.* The anti-patterns *"and" in test name* and *enumerated test names have similar, but different functionality* did have this high correlation between behavior and test name, but the exhibited behavior and names respectively are viewed negatively.

Threats to Validity

This section goes over factors that may impact the applicability of the observations to the real world. It is split into 3 sections: construct, internal, and external validity.

6.1 Construct Validity

This goes over challenges faced that validate whether the findings of this study reflect real-world conditions. The main threat here is whether the sample of 12 open-source projects, 3 being written in C# and 9 written in Java, represent real-world conditions. With a total of 457 unit tests being extracted, this sample is considered accurate as this is an appropriate statistical sample with a 95% confidence level and a 4.52% confidence interval. Therefore, the results of this study should be accurate for other open-source projects, but may not correlate to proprietary systems. Another threat is that the analysis and collection of the test data was done solely by the author. This is mitigated through the use of grounded theory in order to take this subjective data and objectify it through coding and memoing. Having multiple people collect and review the data would have helped to mitigate this threat.

6.2 Internal Validity

Internal validity pertains to the uncontrolled factors that interfere with the results of the study. The main threat here is bias from the author in the finding of test naming patterns. This is mitigated through the use of

grounded theory in order to take this subjective data and objectify it through coding and memoing. Having multiple people collect and review the data also would have helped to mitigate this threat. Another threat was the experience of the developer with test naming practices. This was mitigated through the literature review conducted, allowing the author to gain experience with current test naming practices.

6.3 External Validity

The main external threat to validity with this study was that only open source projects publicly available on GitHub were analyzed for this study. These projects are not representative of all projects in the field, but do provide a good base for finding the preliminary trends that can be analyzed in future work. Also, the random selection of test files analyzed in each of the projects runs the risk of not being a representative selection of the test code, which further limits the generalizability.

Conclusion & Future Work

The objective of this work was to find anti-patterns having to do with test method names that may have a negative impact on developer comprehension. In order to do this a grounded theory study was conducted on 12 open-source Java and C# GitHub projects. From this dataset many patterns were discovered to be common throughout the test code. Some of these patterns fit the necessary criteria of anti-patterns that would probably hinder developer comprehension. With the avoidance of these anti-patterns, it is believed that developers will be able to write better test names that can help speed the time to debug errors, as test names will be more comprehensive.

There are many things that can be done in order to improve upon this research. The first is to verify these anti-patterns are correct, both with expert analysis and with a more diverse sample. This would mitigate many of the threats to the accuracy and generalizability of this research. The other potential for future work with these patterns is on improving automatic test naming tools. By following the good practices found and avoiding the use of the anti-patterns discovered, better test names and naming tools can be created.

Bibliography

[1] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on linguistic antipatterns affecting apis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 25–35. IEEE, 2018.

[2] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[3] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[4] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. IEEE, 2019.

[5] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[6] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. Satdbailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022.

[7] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, page 106675, 2021.

[8] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[9] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167:114176, 2021.

[10] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, page e2395, 2021.

[11] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.

[12] Eman Abdullah AlOmar, Philip T Rodriguez, Jordan Bowman, Tianjia Wang, Benjamin Adepoju, Kevin Lopez, Christian Newman, Ali Ouni, and Mohamed Wiem Mkaouer. How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer, 2020.

[13] Eman Abdullah AlOmar, Tianjia Wang, Raut Vaibhavi, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: An empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2021.

[14] Nicolas Anquetil and Timothy C Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON*, volume 98, page 4, 1998.

[15] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2016.

[16] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gael Gueheneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 187–196. IEEE, 2013.

[17] Nemania Borovits, Indika Kumara, Parvathy Krishnan, Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, Damian A Tamburri, and Willem-Jan van den Heuvel. Deepiac: deep learning-based linguistic anti-pattern detection in iac. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, pages 7–12, 2020.

[18] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *icsm*, pages 97–107, 2000.

[19] C Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pages 112–122. IEEE, 1999.

[20] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67, 2017.

[21] Barney G Glaser and Anselm L Strauss. *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017.

[22] Rashina Hoda. Socio-technical grounded theory for software engineering. *arXiv preprint arXiv:2103.14235*, 2021.

[23] Einar W Høst and Bjarte M Østvold. Debugging method names. In *European Conference on Object-Oriented Programming*, pages 294–317. Springer, 2009.

[24] Bin Lin, Csaba Nagy, Gabriele Bavota, Andrian Marcus, and Michele Lanza. On the quality of identifiers in test code. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 204–215. IEEE, 2019.

[25] Licelot Marmolejos, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, pages 1–17, 2021.

[26] Ettore Merlo, Ian McAdam, and Renato De Mori. Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(4):205–244, 2003.

[27] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.

[28] Christian D Newman, Reem S AlSuhaibani, Michael J Decker, Anthony Peruma, Dishant Kaushik, Mohamed Wiem Mkaouer, and Emily Hill. On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software*, 170:110740, 2020.

[29] Christian D Newman, Michael J Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, 2021.

[30] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[31] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah Alomar, Mohamed Wiem Mkaouer, and Christian D Newman. Using grammar patterns to interpret test method name evolution. *arXiv preprint arXiv:2103.09190*, 2021.

[33] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 350–357, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this?

an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):1–43, 2022.

[35] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 120–131, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Jianwei Wu and James Clause. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software*, 168:110639, 2020.

[37] Benwen Zhang, Emily Hill, and James Clause. Automatically generating test templates from test names (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 506–511. IEEE, 2015.

[38] Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 625–636, 2016.

[39] Jingxuan Zhang, Siyuan Liu, Junpeng Luo, Jiahui Liang, and Zhiqiu Huang. Exploring the characteristics of identifiers: A large-scale empirical study on 5,000 open source projects. *IEEE Access*, 8:140607–140620, 2020.