PROVIDING DEVELOPER-APPROVED DESCRIPTIVE NAMES FOR UNIT TESTS

by

Jianwei Wu

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer and Information Sciences

Winter 2023

PROVIDING DEVELOPER-APPROVED DESCRIPTIVE NAMES FOR UNIT TESTS

by

Jianwei Wu

Approved: _____
Weisong Shi, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____
Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____
Louis F. Rossi, Ph.D.
Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

James Clause, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Lori Pollock, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Sunita Chandrasekaran, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Christian Newman, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to all those who have guided and encouraged me throughout my Ph.D. studies, including my parents, friends, and mentors. I would also like to give a special thanks to my advisor, Professor James Clause, for giving me the exciting research opportunities in the past few years. He has always been inspiring in guiding me through various academic topics. And he put enormous efforts into giving constructive comments and suggestions that better my work and me as a computer science researcher.

Thanks to Professor Lori Pollock, Professor Sunita Chandrasekaran, and Professor Christian Newman for serving on my dissertation committee and giving much appreciated and valuable comments. I want to thank all my friends, Benwen Zhang, Chen Huo, Chunbo Song, Daniel Gaston, Wenhao Wu, Muyuan Li, Fanchao Meng, and Ziqing Luo, who enlighten me academically and personally during our memorable time together in University of Delaware. I also want to thank all my previous managers, Cecilia Jiang (Apple), Rob Whaley (Apple), Mike Chow (Meta), who help me dive into and learn from the real, industrial world of software engineering. Last but not least, thank my parents' company and emotional support I feel lucky to always have.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Unit tests are a critical artifact that supports the software development process in several major ways. For example, when a test fails, its name can provide the first step towards comprehending the purpose of the test and fix the potential bug or issue found in software. As an alternative to reading tens or hundreds of lines of code, comprehension of unit tests is the foundation towards modifying existing tests, adding new tests, or removing unnecessary tests. Therefore, having a descriptive name that can summarize the purpose and context of the test is the key to a successful fix towards the potential bug or issue. Unfortunately, unit tests often lack descriptive names, and non-descriptive names prevent developers from accomplishing the first step of unit testing, comprehension of unit tests.

This dissertation presents a novel method for generating descriptive names for unit tests. This method is based on test patterns and the uniqueness of the test to parse the test body and extract the purpose and context of the test to generate a descriptive name for the test. These combine as a natural and intuitive way to learn the information embedded in the test and use it to produce a descriptive name. Existing work tried to solve the naming problem of unit test. However, existing approaches, either fixing current names or generating new names to replace current names, ignored the vastly existing naming rationales that are commonly used by developers. Therefore, their fixed or generated names often failed to meet developers' approval. Conversely, our method is mainly based on quantitative and empirical studies, which highly focused on the need of developers.

The first contribution of this dissertation is a pattern-based approach to detect non-descriptive test names and provide initial guidance to fix them. After these

non-descriptive names are detected and the initial clue about how to fix them is acquired, the next step is to extract the uniqueness of the test as a middle step towards generating descriptive names. The second contribution of this dissertation is to automatically identify the uniqueness of the test. This step laid the foundation of our name generation approach, which was conducted in an independent, empirical study. After the uniqueness of the test is identified, we can further use it to generate descriptive name. Finally, based on the uniqueness of the test, the last contribution is the name generation approach, which can provide descriptive test names that meet developers' needs.

Chapter 1

INTRODUCTION

Unit tests are an important artifact that supports the software development process in several ways. In addition to helping developers ensure the quality of their software by checking for failures [27], they can also serve as an important source of documentation not only for human developers but also for automated software engineering tools (e.g., recent work on fault localization by *Li et al.* uses test name information [74]). For example, when a test fails, its name can provide the first step towards understanding the purpose of the test and ultimately fixing the cause of the observed failure. Similarly, a test's name can help developers decide whether a test should be left alone, modified, or removed in response to changes in the application under test and whether the test should be included in a regression test suite.

In this dissertation work, we believe that test names are "good" if they are descriptive (e.g, they completely and concisely summarize both the scenario and the expected outcome of the test [13]) and "bad" if they are not descriptive. This is because descriptive names: (1) make it easier to tell if some functionality is not being tested—if a behavior is not mentioned in the name of a test, then the behavior is not being tested, (2) help prevent tests that are too large or contain unrelated assertions—if a test cannot be summarized, it likely should be split into multiple tests, and (3) serve as documentation for the class under test—a class's supported functionality can be identified by reading the names of its tests [118].

Unfortunately, unit tests often lack descriptive names [118, 28]. For example, an exploratory study by *Zhang et al.* found that only 9 % of the 213,423 test names they considered were complete (i.e., fully described the body of test) while 62 % were missing some information and 29 % contained no useful information (e.g., tests named

"test") [118]. Poor test names exist due to developers writing non-descriptive or incomplete names. They can also occur due to incomplete code modifications. For example, a developer may modify a test's body but fail to make the corresponding changes to the test's name. Regardless of the cause, non-descriptive test names complicate comprehension tasks and increase the costs and difficulty of software development.

Because non-descriptive test names negatively impact software development, there have been several attempts to address this issue. For example, *Zhang et al.* and *Daka et al.* used static and dynamic analysis, respectively, to extract important expressions from a test's body and natural language processing techniques to transform such expressions into test names [119, 28]. Researchers have also proposed test generation techniques that attempt to give the tests they create meaningful names (e.g., [108, 36, 118]. Unfortunately, while such techniques can be effective in generating descriptive names, they only work in the context of test generation. They can not be applied to generate names for existing tests.

More broadly, there have been attempts to to improve names for general identifiers and methods. For example, *Høst et al.* proposed an approach for Java methods and variables which uses a set of naming rules and related semantics [54], and *Li et al.* provided a learning-based approach to locate software faults using test name information [74]. Moreover, *Allamanis et al.* and *Pradel et al.* used a model-based and a learning-based approach, respectively, to directly suggest better names or find name-based bugs to facilitate improvements [7, 94]. More recent work attempts to generate names that more closely match existing names by using machine learning to create approaches that can map method bodies to method names (e.g., [9, 10]). However, the success of these approaches heavily depend on their training sets and they are not always successful at creating models that match well with human cognition [76]. In addition, these approaches are primarily geared towards general methods and have problems when applied to generating descriptive names for tests. For example, we have found that, due to the similarity of test bodies, these approaches often generate duplicate names for tests in the same class (see [60]). While such names may make

sense in isolation, they are useless in practice as duplicate names are not possible and, even if they were, they would not serve the goal of helping developers comprehend the purposes of their tests.

The goal of our dissertation work is to develop a set of approaches that can address the problem of poor, non-descriptive test names. At a high-level our work is divided into two pieces.

The first piece is to develop a pattern-based approach to detected non-descriptive test names and, if possible, suggest improvements. This approach is based on a mined set of test patterns and can: (1) detect non-descriptive test names by finding information mismatches between the test name and body of a given JUnit test, (2) provide descriptive information that is a summarization of the test's action, predicate, and scenario, and (3) use the descriptive information to facilitate the improvement of non-descriptive test names by suggesting improvements to developers [114]. Unlike existing approaches which were designed to handle general methods, our approach is specific to JUnit tests. The narrower scope allows the approach to take advantage of the highly repetitive structures that exist in both test names and bodies of JUnit tests To evaluate the pattern-based approach, we select a subset of 508 tests from a total of 26,502 tests that are matched with a pattern. From the collected results of the evaluation, we believe the approach is feasible, accurate at identifying the action, predicate, and scenario from tests, and effective at classifying descriptive and non-descriptive test names.

The second piece is to develop an approach to generate descriptive JUnit test names. This piece of work contains three components.

The first component is to conduct an empirical study of naming rationales that investigated whether tests are named after what makes them unique. A total of 440 tests are selected from 11 open-source projects on Github and used as the experimental subjects in this empirical study. The results of the study (1) demonstrate that a majority of tests are named, either wholly or in part, after what makes them unique, and (2) identify additional aspects that influence how tests are named.

The second component is to build an automated approach for extracting the unique attributes of tests. The approach encodes the knowledge gained from the study into a tool that uses a combination of static analysis and formal concept analysis (FCA) to identify unique attributes. Our evaluation of the approach on a set of 920 randomly chosen tests shows that it is effective. The unique attributes it identifies match human judgment about ≈94 % of the time.

The final component builds a new test name generation approach that builds on both results of the empirical study and the approach for extracting the unique attributes of tests. While the results of the empirical study demonstrated that tests are often named after what makes them unique, it also revealed that a test's name is rarely a straightforward a reproduction of its unique attributes. Instead, names appear to be the result of a complex transformation process that can add to, remove from, or modify the unique attributes. The goal of this component is to understand and then automate this process in order to create a test name generation approach that can create names that are not only descriptive but also match existing naming rationales.

1.1 Contributions

The following sections introduce the contributions of this dissertation. There are two main contributions in this dissertation: (1) a pattern-based approach to detect and improve non-descriptive JUnit test names, and (2) a uniqueness-based approach to generate descriptive JUnit test names.

As the first contribution of this dissertation work, we develop a pattern-based approach that can help developers identify and improve non-descriptive JUnit test names. The pattern-based approach contains: (1) an empirical study to learn the common and meaningful test patterns from a large test corpus and how to use them to help developers detect non-descriptive JUnit test names, (2) a pattern-based approach to compare descriptive information between a test's name and body to facilitate the detection and improvement of non-descriptive JUnit test names, and (3) an evaluation to

measure the pattern-based approach with thousands of tests as subjects for its feasibility, accuracy, and effectiveness. Overall, the results of our evaluation are promising and show that the pattern-based approach is feasible, accurate, and effective for detecting and improving non-descriptive JUnit test names.

As the second contribution towards generating descriptive test names, we develop a uniqueness-based approach to generate descriptive JUnit test names. Our intuition is that developers often name unit tests based on what aspects of the test makes the test unique among its siblings. The uniqueness-based approach contains two primary components: identification of unique attributes and generation of uniqueness-based test names. The identification of unique attributes contains: (1) an empirical study to understand existing test naming rationales and confirm our intuition, (2) a novel approach to extract the unique attributes of a given test that make it unique among its siblings, and (3) an evaluation to compare the identified attributes from our approach with human judgment for its feasibility, consistency, and effort. The results of the evaluation demonstrate that the attributes identified by our approach are consistent with human judgment and are more likely to be useful for future name generation approaches with a minimum need of human effort. Because such attributes often serve as the basis for descriptive names, identifying them is an important first step towards improving test name generation approaches. Therefore, based on the unique attributes from the uniqueness-based approaches, we further develop a uniqueness-based name generation approach to generate descriptive JUnit test names that meet developers' approval. The generation of uniqueness-based test names contains: (1) an empirical study to investigate how unique aspects of a test are transformed to a name, (2) a uniqueness-based approach to generate descriptive JUnit test names, and (3) an evaluation to compare the generated JUnit test names from our approach (i.e., at its worst case setting) with the generated from several alternative approaches judged by 5 professional developers. The results of the evaluation demonstrate that the generated names from our approach earn a high level of strong agreement with human judgment, which are similar to the original test names and significantly higher than the alternative

approaches.

1.2 Outline

The dissertation is composed of four chapters.

Chapter 1 introduces the goal and accomplished work of this dissertation and gives an overall description of the each step of research. Chapter 2 presents the related work. Chapter 3 introduces the first piece of our dissertation work, which is the pattern-based approach to detect and improve non-descriptive test names. Chapter 4 introduces the second piece of our dissertation work, which is to generate descriptive JUnit test names for JUnit tests. At the end, Chapter 5 summarizes the dissertation work and discusses further ideas about future work beyond this dissertation.

Chapter 2

RELATED WORK

Existing work related to our proposed research will be discussed in this section. We have organized them into the following categories:

2.1 Detecting Mismatches/Improving Names

There are some existing works that attempt to identify name/implementation mismatches. *Høst et al.*'s work is the most similar to our approach as it attempts to identify several types of naming bugs in general Java methods [54]. Their approach relies on a manually generated rule book that is extracted from the implicit convention between names and implementations in Java programming, which can be utilized to detect name bugs and provide some suggestions for constructing more suitable names. In their previous works, *Høst et al.* already showed that there is a mutual dependency between method names and their associated implementations [53]. Therefore, their approach considered the mismatch between the name and the implementation of its associated method and used the mismatch to fix name bugs, which are both similar to the analytical process and goal of our pattern-based approach. There are two major differences between their work and ours. First, our approach primarily focuses on the test names rather than general method names that often follow a different naming convention. For example, their approach treated the data type of the value in the `return` statement as an essential attribute in their rule book. However, normally for the unit tests, they compared different values using the `assertions` rather than any `return` statement, so the information in those `assertions` will be a crucial part of their test names. Second, instead of using a manually generated rule book, we built

our approach based on the test patterns, and those test patterns were mined from a large test corpus by a semi-automatic process.

Other existing studies tried to access the naming problem in methods/tests by evaluating the quality of them [120, 11, 79, 48, 122, 23, 86, 93, 24]. All of these approaches managed to either provide appropriate guidance to improve non-descriptive method names or generate accurate summaries for Java methods or classes. However, one major limitation of their work is that developers often have a different mindset when naming their unit test from naming general Java methods or classes. All of the limitations prevent their approaches from being directly used for suggesting descriptive test names without some heavy modifications.

2.2 Automated Generation of Test Names

In contrast to the techniques mentioned above that attempt to improve names, there are several approaches that attempt to automatically generate names. *Daka et al.* introduced a test name generation technique that can summarize a series of coverage goals under the associated test suite [28]. They tried to generate descriptive names for a set of automatically generated JUnit tests [37] that are produced by an automated test generation tool named EvoSuite [36]. Their proposed work utilized a set of unique coverage goals to construct unique names for the automatically generated tests. Both their and our works aimed for a similar goal that is to provide uniqueness-based information to construct test names. However, some key differences exist between their work and ours. Rather than the unit tests written by human, *Daka et al.* mainly focused on constructing descriptive names for the automatically generated tests. Our technique is primarily designed for the manually written tests that appear in nearly every existing projects. Second, instead of using a set of extracted coverage goals that does not represent the real test intent (i.e., stated in their paper), we consider every code element that developers could see in a test. For example, our technique considers CUT methods, methods, assertions, parameters, and objects into the process of extracting unique attributes. All of these code elements are directly related to the

action, predicate, and scenario of each test. Therefore, they can formally describe the real test intent for each test in a specific test class.

Schafer et al. proposed an extensible technique that can address the renaming problem of classes, methods, and fields [97]. The goal of their technique is to provide descriptive names for Java code elements (i.e., classes, variables, and etc.) by inverting lookup functions without sacrificing their defined correctness invariant. Nonetheless, there are two limitations of their technique. First, their technique is primarily developed for the general code elements of Java rather than specifically for the unit tests. Second, they used the lookup functions to build their technique instead of using formal concept analysis, which is capable of processing explicit data like humans. Recent work tried to solve the naming problem of unit tests by using machine learning-based approaches [9, 10]. Although using machine learning-based models can accurately summarize information of the tests, their predicted test names often do not fit developer need which is to be unique among its siblings as the pilot study mentioned in [66]. The observation from the pilot study indicates there is a need to develop an automated approach to extract attributes that can uniquely identify a test from its siblings.

In order to provide descriptive names, existing work aimed at provide better names for methods or unit tests by predefined rules, neural probabilistic language model, and natural-language program analysis [97, 54, 7, 119, 28, 10, 9, 36, 108, 14, 107, 15, 12, 58, 68]. However, although those name generation approaches were aiming to automatically provide better names, two limitations do exist for them to become more successful at providing descriptive test names that meet developers' approval. First, all of the approaches combined the detection of what should be in a descriptive name with the name generation process, which reduced their approaches' flexibility and effectiveness. Second, their techniques combine every steps (i.e., generating both name and body) in the naming process and use their own design and definitions to produce the entire unit test. Because of that, their techniques can not be applied to the vast majority of existing unit tests, which are both manually written and maintained by developers. For example, when developing a project, developers (i.e., especially senior

and principle developers) usually have a pre-defined business or testing plan for their project, and changing business or testing plan in the middle of development circle is often not preferred. To address the importance of identifiers and other names in programming, many researchers utilized different approaches to show the importance of naming the identifiers and present different solutions for the naming issue of identifiers [106, 16, 72, 22, 23]. Although their approaches could be modified to generate descriptive test or method names, they focused on more on name-generation rather than providing distinctive information for unit tests to facilitate better naming.

2.3 General Program Analysis/Automated Testing and Debugging

Several researchers proposed their program analysis or automated testing techniques that can help us have a better understanding of the embedded features in unit tests. Some techniques proposed Java method and class stereotypes on the statement level analysis of Java code [80, 46]. Moreover, *Ghafari et al.* tried to extract the focal method under test [40]. To be more focused on unit testing, *Li et al.* constructed a series of tags for distinguish unit test cases [73]. A group of researchers also conducted a series of work related to tagging methods, rename identifiers or classes with stereotypes [30, 31, 29, 98, 4, 91], but their works might also not be applicable for unit tests. From a general perspective of testing, other researchers tried to devise methods that can perform fully automated testing by the targeted event sequence or the environmental dependencies [58, 15]. All of their works performed well under their specific problems in program analysis or automated testing. However, while their works focus more on extracting features from code or automating the testing process rather than the unit tests themselves, we can still use them to improve our pattern-based approach. Furthermore, *Li et al.* proposed a learning-based approach for fault localization and automated debugging with high performance [74], but the goal of their work is primarily to locate software faults for debugging rather than the naming of unit tests. Regardless of the performance of their proposed tools, the goal of their work is primarily to locate software faults for debugging rather than the naming of unit testing, and the

experimental subjects they used is a standard benchmark database of detecting bugs rather than the unit tests from real-world Java projects. Recent work tried to solve the naming problem of unit tests by using machine learning-based approaches [9, 10]. The data of our quantitative analysis shows that tens or hundreds of duplicate test names were generated for each project (i.e., either using [9] or [10]). This observation indicates there is a need to develop a uniqueness-based naming rationale that can be used to generate descriptive (i.e., being unique and informative) test names.

2.4 Natural Language Program Analysis

There are lots of existing works that try to analyze programs from a natural language-based perspective. *Pollock et al.* and *Shepherd et al.* introduced NLPA by illustrating how to apply NLPA in practice and also giving some insights about aspect mining [93, 99, 92]. Their studies showed natural language clues from developers' naming style can be used for improving the effectiveness of software tools. *Abebe et al.* proposed a natural language-based method to parse the identifier names of program elements for extracting concepts from them [1]. Furthermore, some researchers attempted to split identifiers [33, 24, 47, 51], and others managed to expand abbreviations [52, 77, 26]. Even though their works are not alternatives to our approach, we can still use their implemented tools to improve the accuracy of the test patterns.

2.5 Formal Concept Analysis

As a well-developed technique for deriving a concept hierarchy from objects through a set of attributes, formal concept analysis (FCA) is an excellent method to use for analyzing data like the process of human thinking [38, 117]. Moreover, a early survey has already been conducted to discover the possibility of solving software engineering problem by using concept analysis [109]. Many researchers focused on using FCA as an efficient method to provide solutions for various software engineering problems [43, 89, 25, 50, 78, 111, 34, 78], and they used concept analysis to mock how human (i.e., developers) thinks about writing software or trace the thinking inside

existing software. However, none of them focused on applying FCA to unit tests, and their technique paid more attention to use FCA to understand the high-level structure of software rather than improving the naming in unit tests.

2.6 Investigating Developer Focus in Software Development

To refine the primary selective codes with the secondary codes, we looked into a series of work that investigated developer focus in software development. In recent years, a novel study of tracking the eye movement of developers emerged in the effort to understand the behavior of developers in the process of software development. *Rodeghero et al.* conducted an empirical study to discover the patterns of eye movement when developers perform summarization tasks [96]. *Begel et al.* performed a pilot study of capturing the eye movements of multiple developers in the process of code review, which was inspiring for building a automated tool for code review [18]. *Abid et al.* focused on building a mental cognition model to understand how developers understand codes [2]. *Obaidellah et al.* conducted a throughout survey on the usage of the eye-tracking technology in the research of program comprehension [84]. *Abid et al.* presented a eye-tracking study that shows the behavior of developers when summarizing Java methods [3]. *Ioannou et al.* proposed a series of reading pattern by mining the eye-tracking data for comprehending the behavior of developers [55]. All of these techniques provided an important insight for us to choose which of the code elements should be selected as the secondary codes and helped us to have a profound understanding of the focus of actual developers. Nevertheless, their approaches primarily focused on the eye-tracking of developers when reading the general Java methods and classes, not unit tests. Their approaches did work well for their proposed eye-tracking studies, but no formal approach is proposed to extract unique information from tests that can motivate future name generation approaches.

2.7 Recent Techniques on Test/Method Naming

A more recent work used grammar patterns to understand test names and utilized their findings to recommend better names [90]. *Gao et al.* proposed a functional description-based approach to generate better method names [39]. *Kasegn et al.* proposed a spatial locality-based approach that can recommend method names by using identifier names as a concept [69]. *Nguyen et al.* proposed a machine learning approach to check the consistency between the name of a given method and its implementation [69]. These approaches might work well on each specific field. However, some of them did not performed an empirical study or evaluation with developers, and others' evaluation only included arbitrary developers' responses from open-source projects on Github. Because Github is an open-source platform and public to everyone, there is no guarantee that their collected responses would still be valid if reviewed by professional/core developers from the related projects. An empirical evaluation is needed to verify these approaches with professional developers.

Yamanaka et al. proposed a new technique to accurately recommend extract method refactoring by a set of predicted method names [116]. However, their predicted method names were generated by Code2seq, a older version of Code2vec [9, 10]. Based on the performance of Code2vec in Section 4.6, we believe that this approach might not be effective for providing descriptive test names. In a recent empirical study of test generation approaches (Evosuite [36], etc.), developers have no preference towards the generated tests, and there is always some code refactoring needed for them and their names [49]. Based on their findings, a cooperative approach is preferred by developers.

Chapter 3

A PATTERN-BASED APPROACH TO DETECT AND IMPROVE NON-DESCRIPTIVE JUNIT TEST NAMES

As the first piece of our dissertation work, we developed a pattern-based approach that can help developers identify and improve non-descriptive JUnit test names [114].

The pattern-based approach can: (1) detect non-descriptive JUnit test names by finding mismatches between the name and body of a given JUnit test, and (2) provide descriptive information that consists of the main motive of test, the property to be tested in the test, and the prerequisite needed in the test or the object to be tested (see Section 3.1 for details) to facilitate the improvement of non-descriptive JUnit test names.

Unlike existing approaches that suggest improvements, which were designed to handle general methods, our approach is specific to JUnit tests. The narrower scope of the work allows it to take advantage of the highly repetitive structures that exist in both test names and bodies of JUnit tests (see Section 3.1). From a high-level point of view, the approach uses a set of predefined patterns to extract descriptive information from both a test's name and body. This information is then compared to find non-descriptive names (i.e., cases where the name does not accurately summarize the body). When a mismatch is found, the information used by the approach can help developers address the mismatch and improve the quality of the test name.

To assess the pattern-based approach, we implemented it as an IntelliJ IDE plugin. The plugin was then used to carry out an empirical evaluation of the quality of more than 34,000 tests from 10 Java projects. Overall, the results of our evaluation are promising and show that the pattern-based approach is feasible, accurate, and effective.

```java
public void testExecute_ActionExecutionException(){
    activityGraph.setTop(null);
    try {
        action.execute();
        fail("ActionExecutionException expected.");
    } catch (ActionExecutionException e) {
        //good
    }
}
```

(a) Try-catch statement.

```java
public void testEntries(){
    assertEqualIgnoringOrder(getSampleElements(), multimap().entries());
}
```

(b) Single assertion.

Figure 3.1: Example test patterns.

3.1 Empirical Study: Identification of Test Patterns

In order to develop the pattern-based approach, we conducted an empirical study to learn the common and meaningful test patterns and how to use them to help developers detect non-descriptive JUnit test names. We chose a pattern-based approach because unit tests often have similar structures that can be used to identify the purpose of a test from both its name and body. More specifically, patterns can be used to extract: (1) the **action** which is the focus of the test (i.e., what the test is testing), (2) the **predicate** which are the properties that will be checked by the test, and (3) the **scenario** which are the conditions under which the action is being performed or the predicate is evaluated.

As examples of the common structures shared by unit test bodies, consider the code examples shown in Fig. 3.1. Figure 3.1a shows a unit test whose body consists of a `try-catch` statement. The goal of this type of test is to perform the action under an optional scenario and then to check whether the action was successful or not. In the test corpus used for pattern generation in Table 3.1, we found more than 2,800 tests (≈14 %) shared this structure. Because of the regular structure of this type of test, it is possible to automatically extract its purpose in the form of its action, scenario, and predicate. More specifically, the action is the `method invocation` that occurs before

the "fail" statement in the `try` part of the `try-catch` statement and the scenario is the object on which the action is performed. In Fig. 3.1a, the action of the test is "**execute**" and the scenario of the test is "**action**", the object being tested. For another example, Figure 3.1b presents a unit test whose body contains only a single assertion. The goal of this type of test is to compare the result of an action under a required scenario to an expected predicate. Again, this type of test is common: about 1,000 tests in the corpus mentioned above (≈5 %) share this structure. For this type of test, the action and the predicate can be found by looking at the actual (second) and expected (first) arguments to the assertion statement, respectively, and the scenario will again be the object on which the action is invoked. Therefore, in Fig. 3.1b, the action of the test is "**entries**", the predicate is "**getSampleElements**", and the scenario is "**multimap**".

Common patterns among test names can also be seen in the examples in Fig. 3.1. Figure 3.1a shows an example where the test name (ignoring the leading "test") consists of a leading verb separated from the following noun by an underscore. In our corpus, roughly 7,000 (≈35 %) test names shared this structure. For this structure, the action of the test name is the leading verb and the scenario is the following noun (i.e., in this example, the action is "Execute" and the scenario is "Action").

Similarly, Fig. 3.1b that presents a test name that consists of a single word. In our corpus, about 3,400 (≈17 %) unit tests had one-word test names. In this case, the action of the test name is simply the single word contained in the name (i.e., "Entries" is the action for this example).

In the remainder of this section, we explain the process we used to identify common test name and test body patterns and present a list of the patterns that we used to detect non-descriptive names (see Section 3.2).

3.1.1 Considered Subjects

To identify common patterns among JUnit test names and bodies we considered a set of 18,109 tests comprised of the test suites from the 5 Java projects shown in Table 3.1. These projects are influential open-source projects taken from either

Table 3.1: Considered projects for identifying test patterns.

Project	**Commit Hash**	**# Tests**
Google Guava [45]	473f8d2	14,020
JFreeChart [85]	d03e68a	2,176
JaCoCo [81]	f0102f0	1,323
Weka [113]	d72b95e	436
Barbecue [21]	44a8632	154
	Total	18,109

Table 3.2: Test Body Patterns

Name
If Else
Loop
Try Catch
Try Catch (Restricted)
Try Catch (Generalized)
All Assertion (Single)
All Assertion (Multiple)
Normal (Restricted)
Normal (Generalized)
No Assertion
No Assertion (Generalized)
No Assertion (Specialized for sole method)
No Assertion (Single declaration)
No Assertion (Single method invocation)
No Assertion (Single new object)
No Assertion (Multiple method invocations)
No Assertion (Multiple declarations)

Github [41] or SourceForge [102]. Each project either has thousands of stars on Github (e.g., Google Guava) or has been downloaded more than 10,000 times per week on SourceForge (e.g., Weka). Each project focuses on a different domain: "Guava" is a general-purpose collection of utility classes, "JFreeChart" is a 2D chart library designed for Java applications, "JaCoCo" is a Java code coverage library often used in testing, "Weka" is a machine learning toolkit, and "Barbecue" is used for creating barcodes. Moreover, they are written by different authors so their test suites are likely to have tests written in different ways. Due to these criteria, the patterns we identify from these projects are likely to be general, rather than specific to any one test suite from a project or author.

```
@ITEM=3=}end
@ITEM=0=start{
@ITEM=7=try{
@ITEM=8=catch{
@ITEM=11=}try
@ITEM=10=}catch
@ITEM=15=fail
@ITEM=2=methodCall
@ITEM=-1=*

0 -1 7 -1 8 -1 10 -1 11 -1 3 -1
0 -1 7 -1 2 -1 15 -1 8 -1 10 -1 11 -1 3 -1

start{ * try{ * catch{ * }catch * }try * }end *  1847
start{ * try{ * methodCall * fail * catch{ * }catch * }try * }end *  1470
```

Figure 3.2: Mined Examples by ClaSP.

3.1.2 Test Body Patterns

To identify common body patterns, we used a semi-automated process based on applying frequent pattern mining to the statements contained in test bodies. We chose to operate at the statement-level for two major reasons: (1) statements are the basic syntactic component of tests and standard JUnit tests are composed of statements [67], and (2) while the entire test serves a purpose, individual statements encapsulate sub--steps towards achieving the overall goal [71] such as the action, scenario, and predicate.

The first step in the process was to eliminate inconsequential differences (e.g., literals, variable names, etc.) by abstracting each statement to a number that encodes its type. For example, declaration statements are assigned the number 1, `method invocation` statements are assigned the number 2, etc. While more nuanced abstraction approaches are possible (e.g., def-use-based or graph-based), we found that this approach worked satisfactorily in practice. We also added special symbols to explicitly encode the start and end of each test. These markers are used later to filter out mined patterns that do not span entire tests.

The second step in the process was to apply the ClaSP frequent pattern mining algorithm [44] to the abstracted statements. ClaSP is a novel algorithm that utilized vertical database strategy and heuristic to mine frequent closed patterns. We chose to

use ClaSP because it can efficiently mine the complete set of frequent, closed patterns from its input [35]. This means that ClaSP ensures that each mined pattern has the highest frequency among its super-patterns (i.e., closed, similar to a class and its super-classes), and that long patterns, which are relevant for our purposes, can be mined efficiently. To avoid confusion, we will refer to the output of ClaSP as "proto-patterns" as they serve as the basis for constructing our test body patterns. As the output of this step, we generated 873 proto-patterns.

The final step in the process was to manually examine the proto-patterns to generate test body patterns. This step is necessary because the proto-patterns contain duplicates, spurious entries (i.e., patterns that do not occur in the original tests), and patterns that do not span an entire test. Additionally, since the proto-patterns may contain different setups of where the action, predicate, and scenario should be extracted, we wanted patterns that are both general and that allow for accurately extracting the action, predicate, and scenario.

In Fig. 3.2, the pattern mining process is clearly illustrated by a real-world example that is extracted during the process of pattern mining. The example is composed of three parts: (1) abstracting each statement to a number, (2) using ClaSP to mine frequent patterns from the abstracted statements, and (3) manually examining the proto-patterns to generate test body patterns. First, we utilized an automated script to convert statements to numbers to prepare the corpus for pattern mining, and each type of statement to number pair is also stored for reference (e.g., `methodCall` to 2). Second, after the mining results of ClaSP is completely generated, we collected all generated sequences (e.g., $0\ {-1}\ 7\ {-1}\ 8\ {-1}\ 10\ {-1}\ 11\ {-1}\ 3\ {-1}$) as the proto-patterns and use the statement-number pair from the first step to reconstruct those patterns (e.g., $start\{\star try\{\star catch\{\star\}catch\star\}try\star\}end\star\}$).

Last, we performed a manual examination of the proto-patterns to generate test body patterns. From the last two lines (i.e., each of them is a reconstructed pattern with its number of matches) in Fig. 3.2, although both of them are mined patterns and the first one even has more matches (i.e., 1,847), we only selected the second one as one

kind of representation of the *Try Catch (Restricted)* body pattern shown in Fig. 3.6 since the first one is a spurious entry.

Because the number of proto-patterns is large, we used various grouping strategies to merge similar proto-patterns. In particular, we found that grouping by control-flow statements was effective as such statements often define the high-level structures of test bodies. Another useful approach was to group the proto-patterns by common prefixes in order to identify statement types that were often repeated. The resulting groups of proto-patterns were further examined to eliminate ones that did not include both the special start and end of test markers and ones that did not allow for identifying the action, scenario, and predicate. Finally, the remaining proto-patterns were manually translated in to the 17 test body patterns shown in Table 3.2. For these selected proto-patterns, we manually examined each of them and extracted the action, predicate, and scenario from each pattern by reviewing matched test bodies from those considered projects in Table 3.1. In the remainder of this section each of these patterns will be described in more detail.

```
@Test
public void test...(){
    Class <scenario> = new Class();
    if (<condition related to scenario>){
        <scenario>.<action>;
        ...
    }
    else{
        assert...(...<predicate>);
        ...
    }
}
```

Figure 3.3: If Else.

If Else The motive behind `If Else` body pattern is to capture a type of test body that uses an if-else condition to fulfill its task by testing a particular `method invocation` under a given object in the if part, and use the `assertions` in the else part for its evaluation. As shown in Figure 3.3, the extracted action from this pattern is "⟨*action*⟩" as the first `method invocation` in the `if` part of the `if-else` statement. Then the extracted scenario under test will be the only

"object" that is declared before the `if-else` statement as "⟨*scenario*⟩", and the `method invocation` positioned as the "actual" of the first `assertion` in the `else` part will be the "⟨*predicate*⟩".

```
@Test
public void test...(){
    Class <scenario> = new Class();
    while (condition related to <scenario>){
        <action>;
        ...
        assert...(...<predicate>);
    }
}
```

Figure 3.4: Loop.

Loop The motive of *Loop* body pattern is to include any test body that is trying to repetitively test a `method invocation` under a specific "object", and use its contained `assertion` to evaluate the outcomes.

```
@Test
public void test...(){
    Class <scenario> = new Class();
    try {
        <scenario>.<action>; // Required
        fail(); // Required
    }
    catch (Exception e){
        assert...(...<predicate>); // Optional
    }
}
```

Figure 3.5: Try Catch.

The action in Figure 3.4 is the first `method invocation` inside the `loop` as "⟨*action*⟩", the predicate of the test is the `method invocation` - "⟨*predicate*⟩" positioned as "actual" part of the `assertion`, and the scenario of the test is the "object" used for the `loop` condition as "⟨*scenario*⟩". In addition, the while `loop` is used here as an example, other types of `loop` are also supported.

Try Catch The motive of creating *Try Catch* body pattern is to capture many test bodies that are trying to perform a `method invocation` under a required object and then to check whether the `method invocation` was successful or not. Accordingly, the action of the body in Figure 3.5 is the `method` that was invoked as "⟨*action*⟩". And the object used to invoke the `method` or the leading object declared outside the try-catch statement - "⟨*scenario*⟩" will be the scenario of the test. The `assertion` in the body is *optional*, so there might be no predicate of the test. If there is an `assertion`, then the predicate of the test is the `method invocation` - "⟨*predicate*⟩" positioned in the "actual" part of the `assertion`.

```
@Test
public void test...(){
    ...
    try {
        <scenario>.<action>; // <scenario> is optional
        fail(); // Required
    }
    catch (Exception e){
        assert...(...<predicate>); // Optional
    }
}
```

Figure 3.6: Try Catch (Restricted).

Try Catch (Restricted) The motive of *Try Catch (Restricted)* body pattern is to include a type of test body that is trying to perform a `method invocation` (i.e., action) under an optional object (i.e., scenario) and then to check if the `method invocation` was successfully performed. Accordingly, the action of the body in Figure 3.6 is the method that was invoked as "⟨*action*⟩", and the object used to invoke the `method` - "⟨*scenario*⟩" will be the scenario of the test but it is *optional* for this pattern. The `assertion` is also *optional*, so there might be no predicate of the test. If there is an `assertion`, then the predicate of the test is the `method invocation` - "⟨*predicate*⟩" positioned in the "actual" part of the `assertion`. As we mentioned in Figure 3.1a, that JUnit test is a standard match to the "Try Catch (Restricted)".

```
@Test
public void test...(){
    ...
    Class <scenario> = new Class();
    try {
        ...
        <scenario>.<action>; // Required
        fail(); // Required
    }
    catch (Exception e1){
        assert...(...<predicate>); // Optional
    }
    assert...(...<predicate>); // Optional
    ...
}
```

Figure 3.7: Try Catch (Generalized).

Try Catch (Generalized) This pattern - *Try Catch (Generalized)* body pattern is a more general form of the previous two types of `try-catch` statement-based body patterns. Similarly, the motive of creating this pattern is to capture any test that is trying to perform a `method invocation` under a required object and to check if the `method invocation` was successful. The action and the scenario of the body are in the same places as mentioned in the previous two patterns - "⟨*action*⟩" and "⟨*scenario*⟩" in Figure 3.7. Other statements might appear before the "⟨*scenario*⟩.⟨*action*⟩", but they are considered as "setup" for the action and scenario. The `assertion` for this pattern is still *optional*, but it could appear in the catch part or outside the `try-catch` statement. If there is an `assertion`, then the predicate of the test is the `method invocation` - "⟨*predicate*⟩" positioned in the "actual" part of the `assertion`.

```
@Test
public void test...(){
    assert...(<predicate>, <scenario>.<action>);
}
```

Figure 3.8: All Assertion (Single).

All Assertion (Single) A test body matched by the *All Assertion (Single)* body pattern compares the result of an action under a required scenario to an expected predicate. In *All Assertion (Single)*, the single `assertion` contained

in the test body is trying to compare different results, so the action is the `method invocation` placed in the "actual" position of the `assertion`. The predicate of the test is the `method invocation` placed in the "expected" position of the `assertion`, and the scenario will be the "object" that invokes the action. Therefore, in Fig. 3.8, the action of the body is "⟨*action*⟩", the predicate is "⟨*predicate*⟩" that is required for its comparison and the scenario is "⟨*scenario*⟩", which is also required to invoke the "⟨*action*⟩". Like in Figure 3.1b, the JUnit test is a standard match to the *All Assertion (Single)*.

```
@Test
public void test...(){
    assert...(<predicate>, <scenario>.<action>);
    or
    assert...(<scenario>.<action>((...).<predicate>));
    ...
}
```

Figure 3.9: All Assertion (Multiple).

All Assertion (Multiple) The *All Assertion(Multiple)* body pattern serves as a more general form of the *All Assertion (Single)* pattern. The motive and the locations of action, predicate, and scenario are the same as the *All Assertion (Single)* pattern. There are two differences: (1) the test contains more than one `assertion` as long as they are testing the same action, predicate, or scenario., and (2) there is a new type of `nested method invocation` in the `assertion`. In Fig. 3.9, the action, predicate, and scenario for the first kind of `assertion` are the same as the `assertion` in *All Assertion (Single)*. For the second kind of `assertion`, the action of the body will be the outer `method invocation` as "⟨*action*⟩" that is invoked by the scenario of the body as "⟨*scenario*⟩". The inner `method invocation` - "⟨*predicate*⟩" is the predicate of the body, and it serves as a further step of performing the main action.

Normal (Restricted) The motive of *Normal (Restricted)* body pattern is to capture each test body that tries to perform an action under a scenario as its "setup"

```
@Test
public void test...(){
    Class <scenario> = new Class(<action>);
    anyObject.<scenario>;
    // Declaration or Method invocation or Both
    assert<predicate>(...<action>); // Required
}
```

Figure 3.10: Normal (Restricted).

and evaluate it with a required predicate. The action often appears in the initialization part of the leading `declaration`, but it can also be in the "actual" part of the only `assertion`. The scenario is optional (i.e., none of the first two statements is `method invocation`), but it could be the first `method` being invoked before the final `assertion` or the `object` initialized in the leading `declaration`. The predicate is the `method` name (e.g., "assertEquals", "assertNotNull", etc.) extracted from the `assertion`. In Fig. 3.10, the action of the body is "⟨*action*⟩", the predicate is "⟨*predicate*⟩", and the scenario is "⟨*scenario*⟩".

```
@Test
public void test...(){
    ...
    Class ... = new Class(<action>);
    ...
    anyObject.<scenario>;
    ...
    assert<predicate>(...<action>);
    ...
}
```

Figure 3.11: Normal (Generalized).

Normal (Generalized) The motive of *Normal (Generalized)* body pattern is also to capture a type of test body that tries to perform an action under a specific scenario as its "setup" and evaluate it with a required predicate. This pattern is an extended version of *Normal (Restricted)*, so it shares the similar extraction of the action and predicate. One major difference between this pattern and the previous one is that this pattern allows more statements to be included in the body, and another difference is that this pattern only considers the `method invocation` as the scenario since multiple objects can be declared in the test body. In Fig. 3.11,

the action of the body is "⟨*action*⟩", the predicate is "⟨*predicate*⟩", and the scenario is "⟨*scenario*⟩".

```
@Test
public void test...(){
    Class <scenario> = new Class(...<action>...);
    <scenario>.<action>;
    <...>.<predicate>;
    ...
}
```

Figure 3.12: No Assertion.

No Assertion From a structural perspective, the *No Assertion* body pattern differs from other patterns due to its lack of any `assertion`, which is inspired by one of the test stereotypes mentioned in a recent work [73]. Nonetheless, we greatly extended this type of test body as the following patterns. The motive of *No Assertion* pattern is intended to perform an action under a required scenario, but there is often no primary predicate due to the lack of `assertion`. However, this pattern requires at least three lines of code for its information extraction, and it attempts to extract the predicate of the test. The primary position of the action is in the initialization part of the leading `declaration`, and it can be as the first `method` being invoked after the `declaration`. The scenario is the object that invokes the first `method invocation` (i.e., the action) and is declared in the leading `declaration`. The predicate is the secondary `method invocation` being invoked after the first `method invocation`. In Fig. 3.12, the action of the body is "⟨*action*⟩", the predicate is "⟨*predicate*⟩", and the scenario is "⟨*scenario*⟩".

```
@Test
public void test...(){
    Class <scenario> = new Class(...);
    <...>.<action>;
    ...
}
```

Figure 3.13: No Assertion (Generalized).

No Assertion (Generalized) The motive of *No Assertion (Generalized)* body pattern is also intended to perform an action under a required scenario, but there is often no primary predicate due to the lack of `assertion`. Because the required lines of code decrease to two lines, this pattern is capable of capturing more test bodies. The action changes to the first `method` being invoked after the leading `declaration`. The scenario is the `object` in the leading `declaration`. In Fig. 3.13, the action of is "⟨*action*⟩" and the scenario is "⟨*scenario*⟩".

```
@Test
public void test...(){
    ...
    Class <scenario> = new Class(...);
    <scenario>.<action>;
    ...
}
```

Figure 3.14: No Assertion (Specialized for sole method).

No Assertion (Specialized for sole method) The *No Assertion (Specialized for sole method)* body pattern is a distinctive kind of no assertion-based pattern. The specialization of this pattern is that it only captures a test body with only one `method invocation` across all its statements, which could be an independent `method invocation` or an argument from any statement in the body. In Fig. 3.14, the motive of this pattern is to perform the sole action (i.e., the `method invocation`) under a required scenario, so the action of the body is "⟨*action*⟩" and the scenario of the body is "⟨*scenario*⟩".

```
@Test
public void test...(){
    Class <scenario> = new Class(...<action>...);
}
```

Figure 3.15: No Assertion (Single declaration).

No Assertion (Single declaration) Creating *No Assertion (Single declaration)* body pattern is to include a special kind of no assertion body pattern that can capture any test body with a sole declaration. The motive of this pattern is testing an

"object" that is initialized by a required `method` to check if the "object" can be successfully initialized. In Fig. 3.15, the action of the body is "⟨*action*⟩" as the `method` being invoked, and the scenario of the body is "⟨*scenario*⟩" as the object being tested.

```
@Test
public void test...(){
    <action>(<predicate>...);
}
```

Figure 3.16: No Assertion (Single method invocation).

No Assertion (Single method invocation) Creating *No Assertion (Single method invocation)* body pattern is to include a special kind of no assertion body pattern that can capture any test body with a sole `method invocation`. The motive of this pattern is performing an action that is under a required argument (i.e., predicate) to check if the action can be successfully performed. In Fig. 3.16, the action of the body is "⟨*action*⟩" as the `method` being invoked, and the predicate of the body is the "⟨*predicate*⟩" as the inner argument of the `method invocation`.

```
@Test
public void test...(){
    new <scenario>().<predicate>.<...>.<action>(...);
}
```

Figure 3.17: No Assertion (Single new object).

No Assertion (Single new object) The *No Assertion (Single new object)* body pattern is also a special kind of no assertion body pattern that can capture any test body with a sole new object initialization. The motive of this pattern is initializing a `new object` that is chained to two required `methods` to check if the `new object` can be successfully initialized. In Fig. 3.17, the action of the body is "⟨*action*⟩" as the last `method` being invoked, the predicate is the "⟨*predicate*⟩" as the first `method` being invoked, and the scenario is "⟨*scenario*⟩" as the new object being initialized.

```
@Test
public void test...(){
    Class <scenario> = new Class(...<action>...);
    ...
}
```

Figure 3.18: No Assertion (Multiple declarations).

No Assertion (Multiple declarations) The *No Assertion (Multiple declarations)* body pattern is to create an extension of the *No Assertion (Single declaration)* pattern, and it allows more than one line of code in any captured test body. In Fig. 3.18, the motive and information extraction are the same as the "single method" version: the action of the body is "⟨*action*⟩" as the `method` being invoked, and the scenario is "⟨*scenario*⟩" as the object being tested. Also, the scenario in this pattern needs to be the most frequently evaluated object in the test body, and the action is served as a required argument of that object.

```
@Test
public void test...(){
    <action>(<predicate>...);
    ...
}
```

Figure 3.19: No Assertion (Multiple method invocations).

No Assertion (Multiple method invocations) The *No Assertion (Multiple method invocations)* body pattern is to create an extension of the *No Assertion (Single method invocation)* pattern, and it allows more than one line of code in any captured test body.

The motive and the information extraction are the same as the "single method" version: the action of the body is "⟨*action*⟩" as the `method` being invoked, and the predicate is the "⟨*predicate*⟩" as the inner argument of the `method invocation`, which is also shown in Fig. 3.19. Also, the action in this pattern needs to be the most frequently invoked `method` in the test body, and the predicate is associated with the action as its inner argument.

Table 3.3: Test Name Patterns.

Name
Verb With Multiple Nouns Phrase
Divided Duel Verb Phrase
Is And Past Participle Phrase
Try Catch
Duel Verb Phrase
Noun Phrase
Single Entity
Verb Phrase Without Prepended Test
Verb Phrase With Prepended Test
Regex Match

3.1.3 Test Name Patterns

Because test names are easier to compare since they are shorter than test body we were able to use a fully manual process for identifying commonalities among test names. We found that test names typically fall into two main categories: names that have a common structural format and names that have a common grammatical structure. For the first category, regular expressions can be used directly on the test names to identify relevant pieces of information. For the second category, additional information such as the part of speech of each word in the test name is needed. To obtain this information we used an approach recommended by *Olney et al.* [86]: (1) convert each test name to a sentence by using a purpose-built identifier splitter and prepending the result with the word "I", and (2) apply the Stanford Tagger [104] [86]. The resulting 10 name patterns are shown in Table 3.3 and each is described with more details in the remainder of the section.

```
test<Action><Scenario1><Scenario2><Scenario3>
    POS:Verb   POS:Noun   POS:Noun    POS:Noun
```

Figure 3.20: Verb With Multiple Nouns Phrase.

Verb With Multiple Nouns Phrase This name pattern aims to match a test name that is composed of a prefix of "test" and a verb phrase with multiple nouns. In Fig. 3.20, the first word after the leading "test" should be tagged as "verb" that is the action of the name, and the following three "nouns" are combined as the scenario of the name.

```
test<Action><Scenario><Predicate><Scenario>...
    POS:Verb   POS:Noun   POS:Verb   POS:Noun
```

Figure 3.21: Divided Duel Verb Phrase.

Divided Duel Verb Phrase The motive of this name pattern is to match a type of test names that has a leading "test" followed by a "verb-noun-verb-noun" structure. In Fig. 3.21, the first word tagged as "verb" is the action of the name, and the second word tagged as "noun" is the scenario of the name that is same as the fourth word. The third word should be tagged as "verb" that is the scenario of the name.

```
test<Action><Predicate>...
    POS:Verb POS:Verb, past participle
    (is/are)
```

Figure 3.22: Is And Past Participle Phrase.

Is And Past Participle Phrase This name pattern is intended to match any test name that has a "verb-verb" structure, and the second "verb" should be in its past participle form. In Fig. 3.22, the first "verb" is the action of the name, and the second "verb" is the predicate of the name. If there is a following "noun" after the second "verb", it will be the scenario of the name. However, there were not enough pattern matches to support the scenario, so it is currently not included in this name pattern.

```
test<Action>Throws<Predicate>
```

Figure 3.23: Try Catch (Name).

Try Catch (Name) The "Try Catch" name pattern is designed for test names, and this name pattern belongs to the regular expression-based name patterns. Figure 3.23 shows a more representative sub-pattern of this pattern than other sub-patterns. In the figure, the action of the name is placed before the divider - "Throws", and the predicate of the name is placed after the divider. Moreover, this name pattern is often related to a `try-catch` condition that will be tested in the test.

```
test<Action><Predicate><Scenario>
    POS:Verb  POS:Verb    POS:Noun
```

Figure 3.24: Duel Verb Phrase.

Duel Verb Phrase This name pattern aims to match a type of test names that has a "verb-verb-noun" structure. In Fig. 3.24, the action of the name is the first word tagged as "verb", the predicate is the second word also tagged as "verb", and the scenario is the third word tagged as "noun".

```
test<Scenario>
    POS:Noun
```

Figure 3.25: Noun Phrase.

Noun Phrase This name pattern is set to match any test name that only has one word tagged as a "noun". As shown in Fig. 3.25, the only "noun" is the scenario of the name, and there is often no action or predicate in the name.

```
test<Action (Method Under Test)>
```

Figure 3.26: Single Entity.

Single Entity The "Single Entity" name pattern also belongs to the regular expression-based name patterns, and a representative sub-pattern is shown in Fig. 3.26. After the leading "test", the combination of all following words is the action of the name. Nonetheless, the action of the name needs to fulfill a special requirement that requires the action to be matched to one of the "method under test" [119]. A "method under test" is a `method` that is being tested in the test or the test class. When the action of the name is matched to a "method under test" (i.e., identical names), it will be counted as a pattern match to this name pattern.

```
<Action><Scenario><Predicate>...
POS:Verb  POS:Noun   POS:Verb
```

Figure 3.27: Verb Phrase Without Prepended Test.

Verb Phrase Without Prepended Test This name pattern aims to match any test name that is a "verb phrase" without a prepended word - "test". The "verb phrase" consists of a leading "verb" with a following "noun", and there is a secondary "verb" (i.e., optional) that comes after the "noun". In Fig. 3.27, the action of the name is the leading "verb", the predicate of the name is the secondary "verb", and the scenario of the name is the "noun" between the action and the predicate.

```
test<Action><Scenario><Predicate>...
    POS:Verb  POS:Noun   POS:Verb
```

Figure 3.28: Verb Phrase With Prepended Test.

Verb Phrase With Prepended Test This name pattern aims to match any test name that is a "verb phrase" with a prepended word - "test". The "verb phrase" also consists of a leading "verb" with a following "noun", and there is a secondary "verb" (i.e., optional) that comes after the "noun". In Fig. 3.28, the action of the

name is the leading "verb", the predicate of the name is the secondary "verb", and the scenario of the name is the "noun" between the action and the predicate.

```
when<Scenario>Should<Predicate>
test<Action>After<Scenario>
should<Predicate>If<Scenario>
...
```

Figure 3.29: Regex Match.

Regex Match This name pattern is a collection of 70 regular expression-based sub-patterns. For example, three of the most representative sub-patterns are shown in Fig. 3.29. The first two sub-patterns show a special condition that need to perform the predicate under the defined scenario or execute the action after the scenario is performed. The last sub-pattern is to execute the predicate while the right scenario is successfully performed, and a pattern match in practice is shown in Fig. 3.33a.

3.2 Approach: Detection of Non-descriptive JUnit Test Names

Figure 3.30 presents a high-level overview of our pattern-based approach for detecting non-descriptive test names. As the figure shows, the approach takes as input a JUnit test comprised of its name and body. It then assesses the descriptiveness of the test's name using two phases. The first phase, *pattern-based analysis*, uses the test patterns described in Section 3.1 to extract descriptive information from both the test name and the test body. The second phase, *information comparison*, compares the descriptive information extracted from the name and body against each other. This information comparison process allows for not only detecting non-descriptive test names (i.e., mismatches between the information), but also in some cases indicating to developers how the name could be improved. The remainder of this section describes the two steps of the approach in more detail.

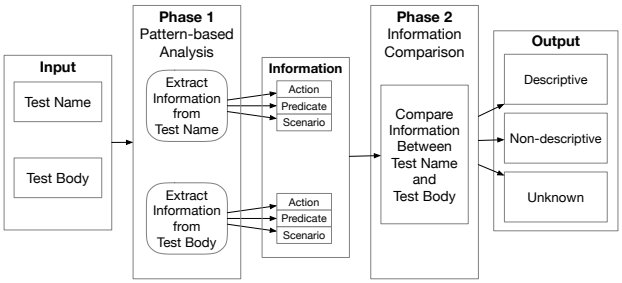


Figure 3.30: Overview of the pattern-based approach.

3.2.1 Phase 1: Pattern-based Analysis

The first phase of the approach is relatively straightforward as it consists mainly of applying the patterns described in Sections 3.1.2 and 3.1.3 to the provided test body and name. If a pattern matches against a name or body, the values it extracts as the action, predicate, and scenario are passed as input to the second stage. If none of the name patterns match or none of the body patterns match, empty values are passed instead.

Generally, if a name or body meets all requirements of a name/body pattern, the name or body is counted as a match to that name/body pattern. In Section 3.1, the requirements of matching each test pattern is stated in a pattern-by-pattern style. For an example of how to match the name pattern, the "Noun Phrase" name pattern can be matched to a test name that is only composed of a leading "test" and an ending noun (i.e., requirements are fulfilled, and the name is considered to be a match to the "Noun Phrase" name pattern), and the ending noun is extracted from the name as the scenario of the name. For an example of how to match the body pattern, the "All Assertion (Single)" body pattern can be matched to a test body that only contains a single and complete JUnit `assertion` (i.e., requirements are fulfilled, and the body is considered to be a match to the "All Assertion (Single)" body pattern). The `expected` part of the `assertion` is extracted from the body as the predicate of the body, and the `actual` part of the `assertion` should be a complete `method invocation` that contains an `object` and a `method call`. The `object` in the `method invocation` is extracted as the scenario of the body, and the `method call` is extracted as the action of the body.

```
public void testGetSSLProtocol() {
    Http11Nio2Protocol protocol = new Http11Nio2Protocol();
    assertNotNull(protocol.getSSLProtocol());
}
```

(a) Example test that is matched by more than one name pattern and more than one body pattern.

	Single Entity	Verb Phrase With Prepended Test
action	GetSSLProtocol	Get
predicate	—	—
scenario	—	SSL

(b) Comparison of information extracted by both matching name patterns for the test shown in Fig. 3.31a.

	Normal (Restricted)	Normal (Generalized)
action	getSSLProtocol	getSSLProtocol
predicate	assertNotNull	assertNotNull
scenario	protocol	—

(c) Comparison of information extracted by both matching body patterns for the test shown in Fig. 3.31a.

Figure 3.31: Example to illustrate the ordering patterns is necessary.

After a match is found, the matched name/body pattern can extract the components from the name/body by using the corresponding positions of the action, predicate, and scenario. Similar to the two examples of matching a pattern to a test name or body that we already mentioned, the extraction process is also straightforward. For the same example of the "Noun Phrase" name pattern, the approach can automatically parse the test name with part-of-speech tags and stores every word in the name with its original order and part-of-speech tag. Then the approach first rules out irrelevant test names (e.g., test names that contain more than two words) and then extracts the first and only noun in the name to be the scenario of the name For the same example of the "All Assertion (Single)" body pattern, the approach is also able to automatically parse code from the statement-level and identifies different types of statements. Therefore, the approach first rules out any test body with more than one statement or contains any kind of statement other than JUnit `assertion`, and it then parse the

JUnit `assertion` to extract the `expected` part and the `actual` part. After all parts of the `assertion` are gathered, the approach extracts the `expected` part (i.e., which is often a `method invocation`) as the predicate of the body, the `object` in the `actual` part as the scenario of the body, and the `method call` in the `actual` part as the action of the body. When every component is successfully extracted from both the test name and body, the approach will determine if the name is descriptive or non-description and generate a report for the associated test as shown Section 3.2.2.

The main complexity in this phase arises from the fact that more than one pattern may match a name or body. For example, Fig. 3.31a shows a JUnit test that can be matched by more than one pattern. More specifically, the test's body can be matched by both the restricted and generalized versions of the "Normal" pattern and the test's name can be matched by both the "Single Entity" and "Verb Phrase With Prepended Test" patterns.

While more than one pattern may match the same name or body, there is often one pattern that is preferred either because it is more accurate at extracting information or it can extract more information. For example, the difference in information extracted by matching patterns can be seen in Figs. 3.31b and 3.31c. Each of these figures, the rows show the values extracted as the action, predicate, and scenario for the patterns shown in the corresponding columns. A dash (—) indicates an empty value that occurs when a pattern did not extract a value for the corresponding type of information. Figure 3.31b is an example of when one pattern may be more accurate at extracting information. In this case, the "Single Entity" pattern correctly extracts "GetSSLProtocol" as the action and does not extract a value for the predicate or scenario while the "Verb Phrase With Prepended Test" pattern incorrectly identifies the action and scenario (i.e., "Get" is tagged as "verb" for the action and "SSL" is tagged as "noun" for the scenario). Figure 3.31c is an example of when one pattern may extract more information. In this case, both the "Normal (Restricted)" and "Normal (Generalized)" patterns correctly identify the action as "getSslProtocol" and the predicate as "assertNotNull" but only the "Normal (Restricted)" pattern identifies the

scenario as "protocol". Because of this difference in performance, it is important to order the patterns to produce the best results.

The ordering of both name and body patterns in our approach is based on our understanding of the patterns, the intuition that more specific patterns should be tried before more general patterns, and the results of applying them to the applications are shown in Table 3.1 as a pilot study. In this pilot study, we tested ten different arrangements of the patterns and selected the one that produced the most accurate ordering. More details about this evaluation process can be find in Sections 3.3.2 and 3.3.3. The resulting orders for the name patterns and body patterns are shown in Tables 3.3 and 3.6, respectively.

3.2.2 Phase 2: Information Comparison

The goal of the information comparison phase is to detect non-descriptive test names. Our approach fulfills this goal by comparing the information extracted from the test name and body. The result of this comparison is that a test name is either: (1) Descriptive, (2) Non-descriptive, or (3) Unknown.

More specifically, each piece of information extracted from a test's name is compared with its corresponding piece of information extracted from the test's body (i.e., $\text{action}_{\text{name}}$ with $\text{action}_{\text{body}}$, $\text{predicate}_{\text{name}}$ with $\text{predicate}_{\text{body}}$, and $\text{scenario}_{\text{name}}$ with $\text{scenario}_{\text{body}}$). If the action, predicate, and scenario extracted from the name are all empty and/or the action, predicate, and scenario extracted from the body are all empty, the name is characterized as Unknown. In this case, it is impossible to determine the quality of the name because an insufficient amount of information was extracted from the name or body.

If there is sufficient information to compare, the approach checks each existing piece of information from the name against the corresponding information from the body. If all of the existing pieces of information match, then the name is considered *descriptive*. Also, if all of the existing pieces from the name is a valid subset of the pieces from the body, the name is still considered *descriptive*. Figure 3.32 shows an example

```
public void testGetGraphNode() {
    GraphNode gn = new GraphNode();
    node = new MyNode(gn, new Point(1, 100), new Rectangle(1, 2, 10, 100),
        Helper.getCollection());
    assertEquals("Equal is expected.", gn, node.getGraphNode());
}
```

(a) Example test with a descriptive name.

	Name	Body
action	GetGraphNode	getGraphNode()
predicate	—	assertEquals()
scenario	—	—

(b) Extracted information for the test shown in Fig. 3.32a.

Figure 3.32: Example of a descriptive test name.

```
public void shouldThrowExceptionIfTokenIsAbsent() {
    final String response = "&expires=5108";
    extractor.extract(ok(response));
}
```

(a) Example test with a non-descriptive name.

	Name	Body
action	—	extract()
predicate	ThrowException	—
scenario	TokenIsAbsent	response

(b) Extracted information for the test shown in Fig. 3.33a.

Figure 3.33: Examples of a non-descriptive test name.

of a test name that is classified as descriptive. The top of the figure shows the test under consideration and the bottom presents a table showing the information extracted by the first phase of the approach. The rows of the table show the values extracted by the pattern type shown in the corresponding column (i.e., the name pattern identified "GetGraphNode" as the action and the body pattern identified "getGraphNode()" as the action). In this example, the name is considered descriptive because all of the non-empty information types match their counterpart (i.e., "GetGraphNode" matches "getGraphNode()").

If, when comparing the name information against the body information, at least

one of the existing pieces of information does not match, then the name is considered *non-descriptive*. It means that a subset of the following conditions happens for that name: (1) $\text{action}_{\text{name}}$ does not match $\text{action}_{\text{body}}$, (2) $\text{predicate}_{\text{name}}$ does not match $\text{predicate}_{\text{body}}$, or (3) $\text{scenario}_{\text{name}}$ does not match $\text{scenario}_{\text{body}}$. Figure 3.33 shows an example of name that is classified as non-descriptive. Again, the top of the figure shows the test under consideration and the bottom presents a table showing the information extracted by the first phase of the approach. In this example, the name is considered non-descriptive because none of the non-empty information types match their counterpart (i.e., "TokenIsAbsent" fails to match "response").

If the outcome is either descriptive or non-descriptive, the approach can sometimes provide additional information to developers to help them improve the test name. For both descriptive and non-descriptive names, if a value provided by the name pattern is empty but the corresponding information provided by the body is not empty, the name can likely be improved by the addition of the body information. For example, Fig. 3.32b shows a test name that is descriptive but can also be improved. In this example, the name accurately reflects that the action of the test is "GetGraphNode" but it is missing information about the predicate that can be found in the body. Adding information that the predicate is "assertEquals" to the name would improve its descriptiveness.

For only non-descriptive names, the approach can suggest modification in two cases. First, if a value provided by the name pattern exists but the corresponding value provided by the body pattern does not exist, the approach suggests that the name information from the name be removed as it is not supported by the body. Second, if the corresponding values provided by the name and body patterns both exist but do not match, the approach can suggest that the information from the name be replaced by the information from the body. For example, Fig. 3.33b shows a non-descriptive test name, which the approach can provide the following suggestions for improvement: First, the predicate part of the name, "ThrowException", should be removed and second, the scenario identified by the name, "TokenIsAbsent", should

also be replaced with the scenario from the body, "response". Note that, because the action from the name is empty, the action identified by the body, "extract", should be added to the name, as described above.

The challenging part of this phase is determining whether the corresponding pieces of information match. Because the information extracted from the name is text while the information extracted from the body are code elements (i.e., `methods`, `objects`, etc.) they can not be directly compared. To address the challenge, the approach automatically converts the *name* of any `method`, `object`, `new instance`, or `assertion method` to a string. For example, the "Normal (Restricted)" body patterns can extract the name of the `assertion method` in Fig. 3.31a, and it is converted to a string that is shown as "assertNotNull" in Fig. 3.31c. Once both the information from the name and the information from the body have been converted to strings, they are also converted to lower case.

The two strings are equal, or if one is strictly contained in the other (i.e., one of them may contain additional information), they are considered to match. Otherwise, they are unmatched.

After we sorted out the process of determining a match, the pattern-based approach can automatically classify each test name in a project as a descriptive name or a non-descriptive name. In the first step, the approach gathers all JUnit tests from the test suite of a selected project by using an automated project analyzer and finds pattern matches for their test names and bodies. In the second step, the approach then uses the test patterns to extract the action, predicate, and scenario from the name and body of each test and generate a report for each name, which is the same as the reports shown in Fig. 3.34. In the last step, the approach automatically aggregates all generated reports for all extracted tests in a comprehensive report for the project, which contains all the detected descriptive and non-descriptive test names. To provide a more intuitive presentation of the approach, an implementation of the pattern-based approach is provided as an IDE plugin [61].

3.3 Evaluation: Detection of Non-descriptive JUnit Test Names

The overall goal of this evaluation is to determine if our pattern-based approach can classify descriptive and non-descriptive test names. However, because the approach's success for this task depends on the underlying patterns, we also evaluate several aspects of their performance. More specifically, we considered the following three research questions:

RQ1—Feasibility. How many test names and bodies are matched by the patterns used by the approach?

RQ2—Accuracy. How accurate are the patterns at extracting the action, scenario, and predicate from test names and bodies?

RQ3—Effectiveness. Can our pattern-based approach correctly classify descriptive and non-descriptive test names?

To investigate these questions, we implemented our approach as an IntelliJ IDEA Plugin [59]. We chose to use IntelliJ IDEA because it is a full-featured IDE that can import projects which use a wide variety of build systems (e.g., Maven and Gradle). This gives us more flexibility in choosing applications when building our set of considered subjects. It also has support for automatically identifying test suites, which are the input to the approach. Finally, it has a robust parsing API that we can use to implement the body patterns. Currently developers can use the plugin by selecting a menu item that analyzes all tests in the current project. In future work, the plugin could easily be extended to support other interaction mechanisms. For example, checking only the names of test in a specific class or the names of individually selected tests.

To generate the experimental data for investigating our research questions, we instrumented the plugin to record the information necessary for answering each research question. We then manually ran the plugin on each of the considered subjects. For each run, the plugin automatically imports the project that is going to be evaluated.

Table 3.4: Considered Subjects.

Project	**Commit**	**# Tests**
Xodus	8d82ef7	940
mytcuml	0786c55	21,532
wheels	15696da	811
EventBus	2e7c046	124
Picasso	5c05678	336
Jenkins	6c1d61a	2,245
ScribeJava	fce41f9	109
mockito	2204944	2,112
Guice	6f1c6cc	1,322
fastjson	4c7935c	4,821
	Total	34,352

After the importing finishes, the plugin will attempt to match every test pattern on each JUnit test that is contained in that imported project. Finally, the plugin outputs a report for all JUnit tests that are evaluated in the process. In total, we collected all information comparison reports for each of the ten Java projects we used in the evaluation. The machine we used for all experiments was a MacBook Pro (2.7 GHz Intel i5 processor; 16 GB RAM) running macOS High Sierra and Java version 9.0.1. Adding up the time for executing the plugin on each project, the total amount of time is roughly five hours for 34,352 tests. Even though the implementation is unoptimized, the execution time is such that it is feasible to include the approach as part of an off-line build process (e.g., overnight).

The remainder of this section describes our considered subjects, research questions, and experimental results in more detail.

3.3.1 Considered Subjects

As the subjects for the evaluation, we selected a set of 34,352 JUnit tests comprised of the test suites from the 10 Java projects shown in Table 3.4. In the table, the first column, *Project*, shows the name of each project; the second column, *Commit*, shows commit hash of the version of the project that was evaluated; and the last

Table 3.5: Match Rates for Name Patterns (Over All Tests).

Name Pattern	**# Matches**	**(%)**
Verb With Multiple Nouns Phrase	0	0.00
Divided Duel Verb Phrase	0	0.00
Is And Past Participle Phrase	0	0.00
Try Catch	204	0.59
Duel Verb Phrase	331	0.96
Noun Phrase	1,555	4.53
Single Entity	4,794	13.96
Verb Phrase Without Prepended Test	2,578	7.50
Verb Phrase With Prepended Test	9,007	26.22
Regex Match	15,883	46.24
Overall	34,352	100.00

column, # *Tests*, shows the number of JUnit tests contained in each project's test suite.

We chose these projects for several reasons. First, they are distinct from the applications and test suites we used to develop the patterns (see Section 3.1). Clearly, the patterns should perform well on the tests that they were derived from. Having separate test suites allows for a more representative evaluation of the first two research questions. Second, the applications they test are diverse since they cover a wide variety of application domains. For example, "Xodus" is a transactional database, "mytcuml" is a UML tool, "wheels" is a testing framework, and "EventBus" is a publish/subscribe pattern-based library that can simplify Android and Java code. In addition, they were written by different developers and at different times. This means that the test suites are not limited to a particular set of authors or patterns and are more likely to be representative than any test from a single project or style. Finally, in aggregate, they have a sufficient number of tests to allow for a thorough evaluation of the approach.

3.3.2 RQ1: Feasibility

The purpose of the first research question is to evaluate the feasibility of our pattern-based approach. The primary way in which we judge feasibility is to determine

Table 3.6: Match Rates for Body Patterns (Over All Tests).

Body Pattern	**# Matches**	**(%)**
If Else	17	0.05
Loop	533	1.55
All Assertion	1,801	5.24
No Assertion	3,602	10.49
Try Catch	5,075	14.77
Normal (Restricted)	1,634	4.76
Normal (Generalized)	13,840	40.29
Overall	26,502	77.15

the percentage of test names and bodies that are matched by one of the patterns used by the approach. In some sense, this is the "coverage" of the patterns. If the coverage of the patterns is low, the usefulness of the approach will also be low as the approach will only be able to provide feedback for a small number of tests. Conversely, if the coverage of the patterns is high, the approach is potentially more useful as it can provide feedback for more tests. However, there is a potential trade-off between the coverage of the patterns and their accuracy (see Section 3.3.3) in that increasing coverage may result in lower accuracy. As such, the sweet-spot for the approach is achieving enough coverage to enable providing feedback for most tests, but not compromising the accuracy of the extracted information.

Tables 3.5 and 3.6 show the experimental data for this research question. In each table, the first column, *Name/Body Pattern*, shows the name of each pattern; the final two columns, *# Matches* and *%*, show the number of times the pattern matched against a test both as a count and a percentage, respectively; and the final row shows an overall summary of the results. For example, the fourth row of Table 3.5 shows that "Try Catch" matched 204 test names (i.e., ≈0.6 % of the 34,352 considered test names). Similarly, the first row of Table 3.6 shows that "If Else" matched 17 of the 34,352 considered test bodies (i.e., ≈0.05 % of the 34,352 considered test bodies). Note that, to simplify the tables and the discussion, most variations of a pattern are grouped into a single row. For example, in Table 3.6, "All Assertion" includes both the "Single"

Table 3.7: Accuracy Results for Each Name Pattern.

Name Pattern	Action (%)		Predicate (%)		Scenario (%)	
	TP	**FP**	**TP**	**FP**	**TP**	**FP**
Verb With Multiple Nouns Phrase	—	—	—	—	—	—
Divided Duel Verb Phrase	—	—	—	—	—	—
Is And Past Participle Phrase	—	—	—	—	—	—
Try Catch	89	11	96	4	89	11
Duel Verb Phrase	96	4	88	12	84	16
Noun Phrase	100	0	100	0	100	0
Single Entity	97	3	97	3	89	11
Verb Phrase With Prepended Test	87	13	74	26	95	5
Verb Phrase Without Prepended Test	100	0	75	25	75	25
Regex Match	84	16	84	16	72	28
Overall	92	8	87	13	89	11

and the "Multiple" versions presented in Table 3.2.

As the final row in each table shows, the overall match rate for both name and body patterns is high. In aggregate, the name patterns matched 34,352 test names (i.e., 100 %), and the body patterns matched 26,502 test bodies (i.e., ≈77 %). While there are a few patterns that had low or zero match rates (e.g., "Divided Duel Verb Phrase"), the cost of keeping such patterns is low as their execution times are low and they may be more prevalent in other project types. The data also demonstrate that the ordering of the patterns is effective. More general patterns (i.e., ones have shown lower in the tables) have higher match rates than more specific patterns (ones shown higher in the tables). Overall, we believe that these results suggest that the approach is feasible. Based on the performance of several related approaches [54, 53, 121, 100, 6], we believe that the coverage of the patterns is high enough to enable the approach to provide feedback for a majority of tests.

3.3.3 RQ2: Accuracy

The goal of the second research question is to investigate whether the patterns can accurately extract information from test names and bodies. Because assessing the accuracy of the extracted information must be done manually (i.e., inspect each test case with the information manually and check if it can correctly describe the action/

Table 3.8: Accuracy Results for Each Body Pattern.

	Action (%)		Predicate (%)		Scenario (%)	
Body Pattern	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
If Else	91	9	36	64	100	0
Loop	89	11	86	14	94	6
All Assertion	100	0	89	11	100	0
No Assertion	96	4	74	26	100	0
Try Catch	100	0	94	6	91	9
Normal (Restricted)	100	0	100	0	100	0
Normal (Generalized)	82	18	100	0	96	4
Overall	94	6	88	12	97	3

predicate/scenario of the name or body by our researchers), it is infeasible to consider all 26,502 tests that were matched by a pattern. Therefore, we chose a subset of information extracted from matched tests to classify. For each name pattern and each body pattern, we randomly selected up to 5 tests matched by that pattern from each project. If no test was matched by that pattern in a project, we skipped the project and moved on to the next one. In total, 242 tests were selected for the name patterns and 266 tests were selected for the body patterns.

For each test in the selected subset, each author manually examined the information extracted by the matching name and body patterns independently. If the extracted information matched the human's judgment it was considered a true positive (TP) and if the extracted information did not match the human's judgment it was considered a false positive (FP). Disagreements among the raters were discussed until a resolution was reached. In total, 1,524 comparisons were made by each rater (i.e., (242 tests for name patterns + 266 tests for body patterns) * 3 comparisons, the action, predicate, and scenario for each test).

Tables 3.7 and 3.8 show the experimental data for this research question. Table 3.7 shows the accuracy of the name patterns and Table 3.8 shows the accuracy of the body patterns. In each table, the first column is the name of each pattern, and the following three pairs of columns show the TP and FP rates for the information

Table 3.9: Effectiveness of the approach.

Project	# Reports	Rate (%)	
		TP	FP
Xodus	29	97	3
mytcuml	105	96	4
wheels	11	91	9
EventBus	10	90	10
Picasso	14	93	7
Jenkins	20	90	10
ScribeJava	3	100	0
mockito	11	82	18
Guice	16	94	6
fastjson	46	98	2
Overall	265	95	5

extracted as the action, predicate, and scenario by each pattern. The final row shows the overall rates for all patterns. For example, the fourth row of Table 3.7 shows the accuracy results for the "Try Catch" name pattern: the TP rate for the action is 89 %, the TP rate for the predicate is 96 %, and the TP rate for the scenario is 89 %. Note that in Table 3.7 a dash (—) indicates the cases where a manual assessment was impossible because the patterns did not match any tests.

The data shown in Table 3.7 and Table 3.8, indicates that the overall accuracy of both the name patterns and body patterns is high. For name patterns, the overall true positive rates range from 87 % for the scenario to 92 % for the action and for the body patterns the overall true positive rates range from 88 % for the predicate to 97 % for the scenario. Even in the worst cases (e.g., identifying the scenario with the Regex Match name pattern), the true positive rate is above 70 %. As such, we believe that both types of patterns are effective at accurately identifying the action, predicate, and scenario from tests.

```
CovariantOverrideTest.returnFoo2
Action:    n/a != return
Predicate: assertEquals != n/a
Scenario:  thenReturn != foo2
-> Non-descriptive
```

(a) A correctly detected non-descriptive test name.

```
SmartNullsStubbingTest.shouldNotThrowSmartNullPointerOnObjectMethods
Action:    tostring != null
Predicate: n/a != null
Scenario:  null == null
-> Non-descriptive
```

(b) A wrongly detected non-descriptive test name.

Figure 3.34: Examples of a true positive and a false positive.

3.3.4 RQ3: Effectiveness

The goal of the third research question is to determine if the pattern-based approach can correctly classify descriptive and non-descriptive test names. Like for RQ2, assessing the output of the approach is a manual process that can not be applied to every output. Therefore, we again selected a representative subset to consider. In this case, because we are interested in the performance across all tests, we chose to consider a total of 265 tests (i.e., 1 % of the 26,502 tests matched by both a name and body pattern). The 265 tests were selected from among each project proportionally to the number of tests in the project's test suite (e.g., 105 tests were taken from "mytcuml", 46 test were taken from "fastjson", etc.).

For each test in the selected subset, each author again manually examined the output of the approach independently. If the output of the approach matched the human's judgment it was considered a true positive (TP) and if the output did not match the human's judgment it was considered a false positive (FP). Disagreements among the raters were discussed until a resolution was reached. The results of the classification are shown in Table 3.9.

In Table 3.9, the first column presents each project's name, the second column shows the number of tests for each project, and the final two columns show the rates

for each classification, respectively. For example, the first row shows there are 29 reports that were selected for project "Xodus", and (≈97 %) of them correctly classify the test name as either descriptive or non-descriptive. The last row shows that the overall TP rate of all reports is ≈95 % (i.e., 251 true positives), and the FP rate is just ≈5 % (i.e., 14 false positives). Owing to the high effectiveness of our pattern-based approach, ≈99.2 % of the 251 true positives are definitively correct. And the definition of correctness here is to be a suitable test name for its related test body. For instance, the test case in Fig. 3.32 is considered to be a true positive since the report can correctly classify its name as a descriptive test name. Additionally, more examples of true positives can be found in the public repository [61]. Because the true positive rate is high with nearly perfect correctness rate, we can conclude that the pattern-based approach is effective at classifying names as either descriptive or non-descriptive.

In addition, we further investigate the two examples in Fig. 3.34 that are selected from the 265 tests from those open-source Java projects. In Fig. 3.34, each example is the output that is produced by our pattern-based approach. The action, predicate, and scenario on the left side of the equations are extracted from the test body, and the action, predicate, and scenario on the right side are extracted from the test name. For the JUnit test in Fig. 3.34a, the test name `returnFoo2` is correctly classified as a non-descriptive test name. Also, some suggestions are provided by our approach for the example in Fig. 3.34a: (1) the action of the name (i.e., `return`) should be removed from the name, and (2) the predicate and scenario of the name should be replaced by the predicate and scenario of the body (i.e., `equals` and `thenReturn`). For the JUnit test in Fig. 3.34b, the test name `shouldNotThrowSmartNullPointerOnObjectMethods` is incorrectly classified as a non-descriptive test name. Because of the difference in length between the short name and long body, the test patterns failed to correctly extract the action, predicate, and scenario from the name and body, and this example is also considered as a false positive.

After we developed the pattern-based approach and evaluated its performance, it is now possible for us to help developers detect non-descriptive JUnit test names

and provide some initial suggestions to improve them. Therefore, the next piece is to generate descriptive JUnit test names to replace the detected non-descriptive names. This next piece of work is described in the following Chapter 4.

Chapter 4

A UNIQUENESS-BASED APPROACH TO GENERATE DESCRIPTIVE JUNIT TEST NAMES

For the second piece of our dissertation work, we developed a new approach to generate test names. Ideally, the names that the approach generates are not only descriptive, but meet with developers' approval. This is important for us because existing work is often not successful at generating descriptive test names with developers' approval. A high-level overview of the research process for this piece of work is shown in Fig. 4.1. This piece of work has two primary components: identification of unique attributes and generation of uniqueness-based test names.

The identification of unique attributes contains three parts in Fig. 4.1. The first part is an empirical study of naming rationales that investigates whether test are named after what makes them unique. The second part is an implementation of a tool that uses what is learned from the study, an automated approach to extract unique attributes of JUnit tests. The first and second parts are published and available online [115]. The last part is an empirical evaluation of the tool using a set of randomly chosen subjects to check if the automated approach can correctly extract unique attributes of tests. As useful as the identification of unique attributes being itself, it is also used to enable the implementation of the generation of uniqueness-based test names.

The generation of uniqueness-based test names also contains three parts in Fig. 4.1. The first part is an empirical study that identifies how to transform the unique aspects of a test into a name. The second part is an implementation of a tool that uses what is learned from the study, a uniqueness-based unit test naming approach. The last part is an empirical evaluation of the tool using a set of randomly chosen subjects to check if the uniqueness-based approach can generate unit test names that

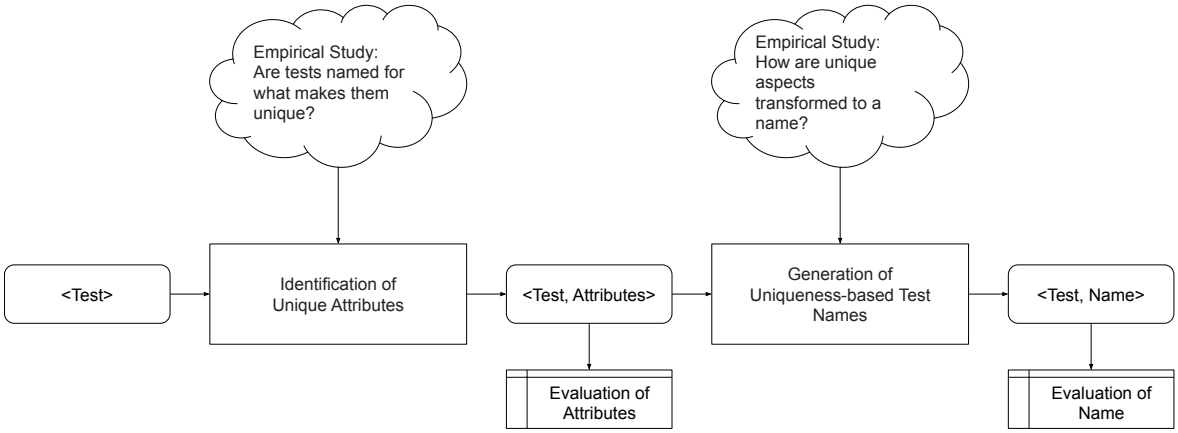


Figure 4.1: Overview of the generation of uniqueness-based names for JUnit tests.

meet developers' needs. The focus of the generation of uniqueness-based test names is more about how to learn from the empirical study of identifying the transformation from the unique aspects of a test to its name and evaluating the generated names of the tool with professional developers.

4.1 Empirical Study: Are Descriptive Tests Named for What Makes Them Unique?

Our belief is that developers often name JUnit tests with descriptive names that are based on what aspects of the test makes the test unique among its siblings. To validate this assumption, we conducted an empirical study of 440 existing tests. First, we examined each test in order to identify what makes it unique from its siblings. Then we examined the test's name and judged whether the name is based on, either entirely or in part, the unique aspects of the test.

4.1.1 Considered Subjects

To gather the tests we examined in our study, we started with the 11 Java projects shown in Table 4.1. In the table, the first column, *Name*, shows the name of the project; the second column, *Version*, shows the version of the project (either

as a Git hash or version number); the third column, *LoC*, shows the number of non-comment, non-blank lines of code as computed by SLOC count [82]; and the final column, # *Tests*, shows the number of JUnit tests in the project. In total, these 11 projects contain 25,459 JUnit tests.

The first ten projects were randomly selected from the top 50 Java projects hosted on Github [56]. Because these projects encompass a variety of domains (e.g., JavaPoet is a library for generating Java programmatically and ExoPlayer is a media player for Android) and have many contributors (e.g., Moshi's test suite contains contributions from 8 different people), their tests are more likely to be representative of tests in general which helps mitigate a potential threat to validity. In addition, we also included Barbecue, a commonly used subject in the testing literature (e.g., [118, 119, 114]). Because Barbecue is significantly smaller than the other subjects, it served as a useful starting point for the study.

Because our investigation is manual, it is necessary to reduce the 25,459 JUnit tests to a more manageable number. Because the projects vary widely in their numbers of tests (e.g., Guava contains 13,962 tests while socket.io-client contains 85 tests), we decided to select a fixed number of tests from each project rather than choosing in proportion to test suite size. This also helps control for threats associated with an unbalanced sample. We found that it took about 5 minutes to understand and encode a single test. Therefore, referencing from similar studies [114, 119], we selected 40 tests from each project, giving us a total of 440 tests; an amount which could be analyzed in around 37 hours (i.e., approximately a week's worth of effort).

When performing the selection of tests, we also had an additional requirement which was to only select tests without poor names. Because we are not interested in generating poor names nor understanding how poor names are chosen (although this may be an interesting area of future work), it makes sense to eliminate tests with non-descriptive names from consideration. Therefore, if a randomly selected test had a name that the authors judged was poor it was replaced with another randomly selected test. We considered a name to be poor if, with the exception of a leading "test", the name

Table 4.1: Experimental Subjects.

Project	Version	LoC	# Tests
Guice	9b371d3	183,049	1,280
Moshi	dbed99d	22,168	716
Picasso	a087d26	11,006	229
Fastjson	e05f1f9	195,511	4,950
Guava	368c337	400,801	13,962
Mockito	22c82dc	59,839	2,145
Socket.io-client	661f1e7	9,478	85
Scribejava	ea42bc9	15,184	110
ExoPlayer	79da521	172,148	1,510
Javapoet	e9460b8	10,755	302
Barbecue	44a8632	10,760	170

(1) contains only numbers (e.g., test12), or (2) contains only numbers, punctuation, and mathematical symbols (e.g., test_1_2). We also made sure not to select any of the 179 tests with empty bodies contained in these applications. The final set of 440 tests is available online, and the complete set of siblings of each chosen test can also be found by its corresponding test class name [66].

4.1.2 Phase 1: Discovering Uniqueness of Tests

The goal of the first part of our study is to identify what makes tests unique. Note that in this work we are assuming that each test has some aspect that makes it unique. While we have observed cases where more than one test has the same body (i.e., duplicate tests with different names), this situation is rare, did not occur among our set of tests, and is likely indicative of a bug as there is no obvious benefit to executing the same test twice.

4.1.2.1 Code Creation Methodology

Identifying what makes a test unique involves comprehending not only the test under consideration but also the test's siblings (i.e., tests that influence or restrict the name of a test). In this study we consider tests declared in the same class as siblings. We chose this granularity for several reasons. First, the Java programming language

```
public void testDrawBarUsingBackgroundColourActuallyDrawsWithBackgroundColour() throws Exception {
    output.drawBar(0, 0, 10, 100, false);        Action
    assertEquals(g.getColor(), bgColour);
}
```

(a) Action.

```
public void testDrawTextRendersString() throws Exception {
 output.drawText("FOO", LabelLayoutFactory.createCenteredLayout(0, 0, 100));
 Rectangle r = g.getTextBounds();
 assertTrue(r.getWidth() > 0);
 assertTrue(r.getHeight() > 0);        Predicate
}
```

(b) Predicate.

```
public void testTextIsNotDrawnIfFontIsNull() throws Exception {
 output = new GraphicsOutput(g, null, fgColour, bgColour);        Scenario
 double height = output.drawText("FOO", LabelLayoutFactory.createCenteredLayout(0, 0, 100));
 assertEquals(0, (int) height);
 assertNull(g.getTextBounds());
}
```

(c) Scenario.

```
public void testEqualsComparesBarWidths() throws Exception {
    Module mod = new Module(new int[] { 2, 2, 2, 1, 2, 4 });
    Module mod2 = new Module(new int[] { 2, 2, 2, 1, 2, 4 });    Scenario+Predicate
    assertEquals(mod, mod2);
}
```

(d) Combination.

Figure 4.2: Examples of primary codes.

forbids methods with the same signature in the same class. Because tests have no parameters, this means that each test (method) in a class must have a unique name. Second, tests in the same class are likely to be related (e.g., they share the same class under test). This means that the aspects that make them unique are more likely to be limited in scope and therefore more interesting with respect to how tests are named.

Because there is no pre-existing classification scheme for identifying what makes a test unique, we used open, axial, and selective coding to qualitatively analyze the tests [42, 105]. First, each author individually examined each of the 440 considered tests and tagged the portions of the test body that they believe are the unique aspects of the test. These portions of the test body are the potential components of the tagged text in the following sections and can be different types of code elements such as method invocations, parameters, and objects. Each tag consisted of a word or short phrase that characterizes the type of uniqueness. After each author tagged each test individually, the authors together examined the tagged tests in order to reach consensus on which portion of a test's body makes it unique and to discuss the open codes. Then the authors used axial coding to establish relationships among the open codes and generated a final list of selective codes.

4.1.2.2 Selective Codes

The set of selective codes is based partly on the Java language elements [87] that can comprise a test (e.g., variable declarations, method calls, control flow structures,

parameters and arguments, etc.). However, we found that it was desirable to both refine these codes and to include other, broader codes, in order to more precisely capture concepts that are specific to JUnit tests. More specifically, we created four primary codes that correspond to the high-level structure of the test (i.e., parts of test [119]) and multiple secondary codes that refer to test-specific elements (e.g., calls to methods of the class under test, expected or actual arguments to assertions, etc.) which are mentioned by existing approaches [40, 28, 114]. Because the secondary codes refine the primary codes, they can only be applied if their corresponding primary code is applied first. Below, we discuss each primary code and its associated secondary codes in detail. Moreover, Fig. 4.2 shows code examples for each of the primary codes.

4.1.2.2.1 Action

The Action code is a primary code that is applied to a test when no sibling shares the test's Action—the part of the test that is the primary interaction with the application under test. The secondary codes for the Action code relate to: (1) the elements of the action, specifically methods calls and arguments, and (2) whether the method calls and arguments are related to the class under test.

Because, in the context of testing, method calls to the class under test are more important than calls to methods declared by other classes and method calls in general are more important than method arguments, the secondary codes are prioritized as shown below; a lower ranked code can only be applied if no high-ranked code has been applied. This ranking is based on our intuition as well as recent studies that use eye-tracking technology to understand the importance of code elements for different software engineering tasks (e.g., [96, 95, 18]).

(1) The Class Under Test (CUT) Method Call code is applied when (i) the test's action contains a call to a method declared by the class under test, and (ii) no other sibling contains a call to the same method in its action, irrespective of the method arguments

(2) The non-CUT (other) Method Call code is applied when (i) the test's action contains a call to a method declared by a class that is not the class under test, and (ii) no other sibling contains a call to the same method in its action, irrespective of the method arguments

(3) The CUT call arguments code is applied when the test's action contains a call to a method declared by the class under test (CUT) that has a unique set of method call arguments

(4) The non-CUT (other) call arguments code is applied when the test's action contains a call to a method declared by a class that is not the class under test that has a unique set of method call arguments.

For example, Fig. 4.2a shows an example of the Action primary code from Barbecue's `GraphicsOutputTest` class, and the colored box (i.e., circled in blue) indicates where we should be looking for in the test body [119]. In this test, the Action code can be applied to its first statement: `output.drawBar(0, 0, 10, 100, false);` as a method invocation, and no other sibling from the same class shares its Action.

4.1.2.2.2 Predicate

The Predicate code is applied to a test when no sibling shares the test's predicate—the part of the test that checks the result of performing the action. The secondary codes for the Predicate code relate to: (1) the assertions used by the test and, and (2) the arguments passed to the assertions. Again, the secondary codes are prioritized based on their relative importance in the context of testing and a lower-ranked code can only be applied if a higher-ranked code has not already been used. The Predicate code has noticeably more secondary codes than other primary codes. First, the definition of unit testing is to test a minimum component of a software, so the testing process is usually involved with checking the produced results. Second, when checking the produced results, there are many different kinds of result-checking statements that can be used in a test, which will make the test unique.

(1) The actual and expected parameters code is applied when (i) the test's predicate contains a pair of actual and expected parameters that is extracted from an assertion call, and (ii) no other sibling has the same pair of actual and expected parameters in its assertion calls

(2) The actual parameter code is applied when (i) the test's predicate contains an actual parameter that is extracted from an assertion call, and (ii) no other sibling has the same actual parameter in its assertion calls

(3) The expected parameter code is applied when (i) the test's predicate contains an expected parameter that is extracted from an assertion call, and (ii) no other sibling has the same expected parameter in its assertion calls

(4) The assertion call code is applied when (i) the test's predicate contains an assertion call that is extracted from the assertions of the test, and (ii) no other sibling has the same assertion call and uses it as its predicate

(5) The other result-checking call code is applied when (i) the test's predicate contains other types of result-checking calls that serve the same functionality as JUnit assertions, and (ii) no other sibling has the same set of result-checking calls and uses it as its predicate

(6) The only assertion code is applied when the test's predicate contains a call to an JUnit assertion method and no other sibling calls any JUnit assertion method (i.e., the only test with JUnit assertion).

For example, Fig. 4.2b shows an example of the Predicate primary code from Barbecue's `GraphicsOutputTest` class, and the colored box indicates where we should be looking for in the test body. In this test, the Predicate code can be applied to its last two statements: `assertTrue(r.getWidth() > 0); assertTrue(r.getHeight() > 0);` as assertions, and no other sibling from the same class shares its Predicate.

```
public void shouldReturnTimestampInSeconds() {
  final String expected = "1000";   Predicate - actual parameter
  assertEquals(expected, service.getTimestampInSeconds());
}                                   Action - method call (CUT)

public void shouldReturnNonce() {
  final String expected = "1042";   Predicate - actual parameter
  assertEquals(expected, service.getNonce());
}                                   Action - method call (CUT)
```

(a) Test cases with tagged text from Scribejava.

```
public void completeSetsBitmapOnRemoteViews() throws Exception {
  RemoteViewsAction action = createAction();
  action.complete(BITMAP_1, NETWORK);   Action - method call (CUT)
  verify(remoteViews).setImageViewBitmap(1, BITMAP_1);
}                                       Predicate - actual parameter

public void errorWithNoResourceIsNoop() throws Exception {
  RemoteViewsAction action = createAction();
  action.error();
  verifyZeroInteractions(remoteViews);
}  Predicate - Other Result-checking Call

public void errorWithResourceSetsResource() throws Exception {
  RemoteViewsAction action = createAction(1);
  action.error();                   Scenario - variable initialization (OUT)
  verify(remoteViews).setImageViewResource(1, 1);
}                                   Predicate - actual parameter
```

(b) Test cases with tagged text from Picasso.

Figure 4.3: Examples of test cases with tagged text.

4.1.2.2.3 Scenario

The Scenario code is applied to a test when no sibling shares the test's scenario—the part of the test that configures or sets up the environment under which the action will be performed.

The secondary codes for the Scenario code relate to: (i) the elements of the scenario, specifically variable initialization and arguments in assertions, (ii) whether the variable initialization and arguments are related to the object under test, and (iii) control flow variable and state-changing CUT call.

Because, in the context of testing, variable initialization to the object under test are more important than variable initialization to other objects and variable initialization in general are more important than variable initialization arguments and control flow variables. And the state-changing CUT method call code is added in the list to meet the possible case of having a focal method call [40]. Therefore, the secondary codes are prioritized as shown below and a lower ranked code can only be applied if no high-ranked code has been applied.

(1) The Object Under Test (OUT) variable initialization code is applied when (i) the

test's scenario contains a variable initialization that is used in a variable declaration for an object under test, and (ii) no other sibling contains the same variable initialization for its variable declaration for OUT and uses it as its scenario, irrespective of the arguments used in the variable initialization. The OUT variable initialization will be the tagged text, and the rest of the secondary code of the Scenario code follows the same rule (i.e., code elements are converted to the tagged text).

(2) The non-OUT (other) variable initialization code is applied when (i) the test's scenario contains a variable initialization that is used in a variable declaration but not for any object under test, and (ii) no other sibling contains the same variable initialization for its variable declaration (i.e., not for OUT) and uses it as its scenario, irrespective of the arguments used in the variable initialization.

(3) The non-OUT (other) variable initialization arguments code is applied when (i) the test's scenario is composed of a set of arguments that are used in the variable initialization of a variable declaration but not for any object under test, and (ii) no other sibling contains the same set of arguments that are used in variable initialization of a non-OUT variable declaration and uses it as its scenario.

(4) The control flow variable code is applied when (i) the test's scenario contains a variable that is used in the conditional statement of a control flow-based statement (i.e., loop, if-else, or other type of statement), and (ii) no other sibling contains the same variable that is used in the same type of control flow-based statement and uses it as its scenario.

(5) The state-changing CUT method call code is applied when (i) the test scenario is composed of a CUT method call that changes the state of the test [40], and (ii) no other sibling contains the same state-changing CUT method call in its scenario, irrespective of the method arguments.

For example, Fig. 4.2c shows an example of the Scenario primary code from Barbecue's `GraphicsOutputTest` class, and the colored box indicates where we should be looking for in the test body. In this test, the Scenario code can be applied to its first statement: `output = new GraphicsOutput(g, null, fgColour, bgColour);` as object initialization, and no other sibling from the same class shares its Scenario.

4.1.2.2.4 Combination

The Combination code is applied to a test when none of the other primary codes is applicable (i.e., the test's action, predicate, and scenario are shared with other tests). The secondary codes for the Combination code enumerate the possible combinations of the first three primary codes: Action and Predicate, Action and Scenario, Scenario and Predicate; Action, Scenario, and Predicate. For example, the Action and Predicate code will be applied when the action and predicate of the tests are not unique on their own but the combination of both of them is unique among other tests in the same test class. The rest of the secondary codes of the Combination code follow the same pattern. The tagged text of each secondary code of the Combination code will be the structure of the combination, such as action and predicate or action and scenario.

For example, Fig. 4.2d shows an example of the Combination primary code from Barbecue's `ModuleTest` class, and the colored box indicates where we should be looking for in the test body. In this test, the Combination code can be applied to its several statements as a specific combination of Scenario and Predicate, and no other sibling from the same class shares this combination of Scenario and Predicate.

4.1.2.3 Coding Process

Using the selective codes and guidelines described above, each author individually recoded all of the 440 tests. This was more straightforward than the initial open coding process because we were now familiar with the test and the results of the open and axial coding processes were already available. Each test was first assigned one or

more of the four primary codes. Then, for each primary code that was assigned, one or more of its associated secondary-codes were assigned.

As an example of the selective coding process, consider the examples in Fig. 4.3 (i.e., codes are highlighted in blue). The top-half of each figure shows an excerpt of a test class from one of the subjects considered in the study. Note that some minor reformatting has been done to improve the presentation (i.e., all spaces and comments are removed). The bottom of each figure shows the codes applied to each test using the format: ⟨`top level code`⟩ - ⟨`secondary code`⟩: ⟨`tagged text`⟩ where `tagged text` shows the portion of the test body that is tagged by the secondary code.

For example, in Fig. 4.3a, the test `shouldReturnTimestampInSeconds` is tagged with Action - method call (CUT): "`getTimestampInSeconds`" because no sibling shares the test's action, which is to call the service's timestamp functionality (i.e., the CUT Method "`getTimestampInSeconds()`"). This test is also tagged with Predicate - actual parameter: "`service.getTimestampInSeconds()`" because no sibling uses the result of calling "`getTimestampInSeconds`" as the actual parameter to an assertion method.

In Fig. 4.3b, the test `completeSetsBitmapOnRemoteViews` is tagged with Action - CUT method call: "`complete`" because no sibling shares the test's action, which is to call a method to the action's complete functionality (i.e., the CUT Method "`complete()`"). This test is also tagged with Predicate - actual parameter: "`setImageViewBitmap(1, BITMAP_1)`" because no sibling utilizes the `setImageViewBitmap(1, BITMAP_1)` as the actual parameter in an assertion. The test `errorWithNoResourceIsNoop` is tagged with Predicate - other result-checking call: "`verifyZeroInteractions`" because no sibling shares the test's predicate, which is to call a verification method `verifyZeroInteractions` to check the behavior of the `remoteViews` variable. The test `errorWithResourceSetsResource` is tagged with Predicate - actual parameter: "`setImageViewResource(1, 1)`" because no sibling shares the test's predicate, which is to verify the behavior of `remoteViews` variable with a specific parameter and utilizes the `setImageViewResource(1, 1)` as the actual parameter in an assertion. This test is also tagged with Scenario - variable initialization (OUT): "`createAction(1)`" because no

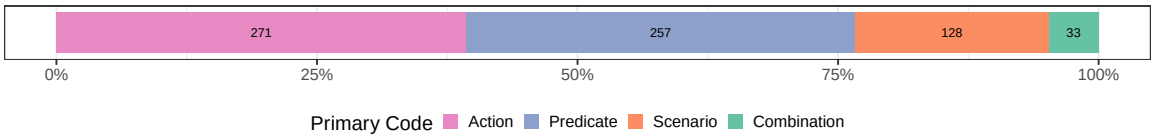


Figure 4.4: Frequency of primary codes.

sibling shares the test's scenario because no other test initializes the "action" variable (i.e., the object under test) using the `createAction(1)` as its variable initialization.

The agreement between the raters was high (Fleiss' $\kappa = 0.92$) which suggests that the guidelines are detailed enough to support a repeatable process. After each author coded each test individually, the authors together examined the tagged tests in order to discuss and address any disagreements in the results. During the process of coding each test, the tagged text was manually extracted from each tagged test by using the coded results to locate the exact code elements (i.e., converted to text) that make the test unique among its siblings in the same class. At the end of this coding process we created a topical concordance that shows, for each code, the tests that were tagged with the code. For the 440 tests that were selected from 11 projects, all the tagged results are shown in the online document [66].

4.1.2.4 Result and Discussion

To better understand the results of the coding process we gathered some descriptive statistics. Figure 4.4 shows the frequency of the primary codes across the sampled tests. In this horizontal stacked bar-chart, the color legend shows the name of each primary code and the stacked four bars, Action through Combination, show the number of times each primary code was applied to tests among the 440 subjects. For example, the first stacked bar shows that, for the tests sampled from the 11 projects, the Action code was applied 271 times, the second shows that the Predicate code was applied 257 times, and the third shows that the Scenario code was applied 128 times, and the Combination code was applied 33 times.

Figure 4.5 shows the frequencies of each secondary code in four horizontal

Figure 4.5: Frequency of secondary codes.

stacked bar-charts. In each plot, the upper text (e.g., Action, Predicate) shows the primary code that each secondary code corresponds to; the color legend, Secondary Code, shows how often the secondary code was applied. For example, the top bar-chart shows that the secondary code "CUT method call" was applied 77 times which is approximately ≈28 % of the 271 times the primary Action code was applied (i.e., when it was possible to apply the secondary code).

Based on the data shown in these plots, we can draw two general observations. The first observation is about the relative frequencies of the top level codes: the Action and Predicate codes are the most common and occur about the same amount of the time (271 out of 689, ≈39 % and 257 out of 689, ≈37 %, respectively); the Scenario code is less common (128 out of 689, ≈19 %); and the Combination code is the least common and is relatively rare (33 out of 689, ≈5 %). This suggests that there is often a single part of a test that makes it unique among its siblings and the action and

predicate are what makes it unique most often. The second observation is about the relative frequencies of the secondary codes. With the exception of the OUT Variable Initialization secondary code (94 out of 128, ≈73 %), there are no dominant secondary codes (e.g., >70 %). This suggests that while there may be trends among the part of a test that makes it unique (e.g., action, scenario, and predicate), the specifics of what make each of these parts of the test unique varies greatly. In addition, with the exception of Javapoet (which relies on the secondary codes under the predicate primary code, 35 out of 59, ≈60 %), there are also no dominant secondary codes (e.g., ≥60 %) at the project level, and the rest of the detailed data can be found in the online document [66].

4.1.3 Phase 2: Deciding Whether JUnit Tests with Descriptive Names are Named After What Makes Them Unique

The goal of the second part of our study is to investigate whether tests with descriptive names are named, either wholly or in part, after what makes them unique. In order to decide whether tests with descriptive names are named after what makes them unique, we investigated two research questions:

- **RQ1**: Is uniqueness used as a naming rationale for JUnit tests?
- **RQ2**: When uniqueness is the naming rationale, does the tagged text appear directly in the name or is it transformed?

4.1.3.1 RQ1: Is Uniqueness Used as a Naming Rationale in JUnit Tests?

To investigate RQ1, we manually compared the name of each test against the aspects that make it unique which were identified in the first part of the study. The comparison between the test name and tagged text (i.e., the aspects of the test that make it unique) was performed as follows. First, we automatically split the test name and the tagged text into a set of tokens using a customized tokenizer for identifiers [33]. Then we convert all tokens to lower case remove any leading "test" as well as connectors that are rarely used in the test body (e.g., not, to, then, etc.) [66]. For

example, the test name `testPostReturnsBarcodeImage` is parsed into a set of tokens: ⟨*post*, *returns*, *barcode*, *image*⟩. Finally, we manually compared the two sets of tokens. Each author did this comparison individually and classified the tests into one of the following categories based on the degree to which the name appears to be based on the tagged text.

- **Full**: The tests in the Full category appear to be named wholly after what makes the test unique (i.e., every token from the name appears to be derived from a token from the tagged text). For example, for `testPutAll` from Guava, the tokens from the name are ⟨*put*, *all*⟩ and the tokens from the tagged text are also ⟨*put*, *all*⟩. Because each token in the token set of the name originates from a corresponding token from the token set of the tagged text (e.g., both tokens are literally the same or share the same meaning), `testPutAll` is included in the Full category. As another example, consider `test_geo` from FastJson. For this test, the tokens from the name are ⟨*geo*⟩ and the tokens from the tagged text are ⟨*geometry*⟩. Because "geo" appears to be an abbreviation for "geometry", each token from the name is derived from a token from the tagged text and the test is also included in the Full category.

- **Partial**: The tests in the Partial category appear to be named partially after what makes the test unique (i.e., at least one token from the name appears to be derived from a token in the tagged text). For example for `testChecksumIsNull` from Barbecue, the tokens from the name are ⟨*checksum*, *is*, *null*⟩ and the tokens from the tagged text are ⟨*calculate*, *checksum*⟩. Because the token "checksum" in the token set of the test name is directly derived from the token "checksum" in the token set of the tagged text, `testChecksumIsNull` is included in the Partial category. As another example, consider `testChildBindingsNotVisibleToParent` from Guice, the tokens from the test name are ⟨*child*, *bindings*, *visible*, *parent*⟩ and the tokens from the tagged text are ⟨*get*, *binding*⟩. Because the token "bindings" from the test name is the plural

Table 4.2: Results to check if the tokens are Identical or Transformed.

Project	Full			Partial			None
	# Tests	Identical	Transformed	# Tests	Identical	Transformed	# Tests
Barbecue	2	0	2	32	27	5	6
Moshi	5	5	0	18	16	2	17
Mockito	4	3	1	27	24	3	9
Guava	2	2	0	30	27	3	8
Guice	1	1	0	36	29	7	3
ExoPlayer	3	3	0	19	16	3	18
Scribejava	5	5	0	31	28	3	4
Socket.io-client	0	0	0	36	22	14	4
Fastjson	1	1	0	20	19	1	19
Picasso	2	2	0	29	25	4	9
Javapoet	0	0	0	31	28	3	9
Overall	25	22	3	309	261	48	106

of the token "binding" from the tagged text, `testChildBindingsNotVisible-ToParent` is also included in the Partial category.

- **None**: The tests in the None category do not appear to be named after what makes the test unique (i.e., none of the tokens from the name appear to be derived from a token from the tagged text). For example, for `testValueIsRequired` from Barbecue, the tokens from the test name are $\langle value,\ is,\ required\rangle$ and the tokens from the tagged text are $\langle remove,\ servlet,\ exception\rangle$. Because none of the tokens from the test name appears to be derived from the tokens from the tagged text, `testValueIsRequired` is listed in the None category.

The agreement between the raters was high (Fleiss' $\kappa = 0.88$) which suggests that the category descriptions are detailed enough to support a repeatable process. Again, after each author classified each test individually, the authors together examined the classification in order to resolve any disagreements.

The results of the rectified classification are shown in Table 4.2. In the table, the first column, *Project*, shows the name of each project and the subsequent three columns, *Full*, *Partial*, and *None*, show the number of tests in each category, respectively. The first sub-column, *# Tests*, indicates the total number of tests that is under each main category and the other two sub-columns, *Identical* and *Transformed*, will be explained

later. For example, the first row shows the data collected from Barbecue: a total of 2 tests are in the Full category, a total of 32 tests are in the Partial category, and a total of 6 tests are in the None category (i.e., sum to 40 per project). Finally, the last row shows the totals for each category across all inspected projects.

As the data in Table 4.2 shows, most of the tests (334 out of 440, ≈76 %) are in either the Full or the Partial category, with Partial being the largest category. However, there are also a significant fraction of tests in the None category (106 out of 440, ≈24 %). We found this surprising, especially considering that tests with bad names were excluded from our set of tests.

To better understand the tests that are not named after what names them unique, we further investigated the tests in the None category. We found that while these test names are not based on what makes the test unique, they do often follow a reasonable naming rationale. The most common rationale (34 out of 106, ≈32 %) is naming a test after something that is out of the scope of test bodies such as the setup of the testing environment or an implicit behavior of the program. For example, `testValueIsRequired` from Barbecue is designed to make sure when there is a missing value in the required parameters of the setup of its test class, an exception will be thrown and caught. The second most common rationale (30 out of 106, ≈28 %) is to name a test after after a user behavior (i.e., during integration testing). For example, `loadThrowsWithInvalidInput` from Picasso is designed to make sure that when the user enters an invalid input (i.e., URL for this test), a corresponding exception must be thrown. The third most common rationale (29 out of 106, ≈27 %) is to name the test after a part of the test that is not unique. For example, `testReadFull` from Exoplayer is designed to check if a specific data source can be fully read. This test is named after the method call $\langle read \rangle$, which is also called by this test's siblings.

Overall, we found that 88 of the tests the None category were named using a reasonable rationale. The remaining tests (13 out of 106, ≈12 %) were tests with poor names that were not caught by our conservative filtering. For example, JavaPoet contains a test named "usage" that does not describe the purpose of the test. In our

planned future work, we will investigate approach to identify these other rationales so that the can be used to generate descriptive names.

Based on the data in Table 4.2 and our additional investigation of tests in the None category, we conclude that the answer to RQ1 is "yes": uniqueness is indeed commonly used as a naming rationale when constructing the names of JUnit tests. From the 440 selected tests, most of them are named under the uniqueness-based naming rationale, and their test names at least partially reflect the uniqueness by containing some or all of the tokens from the tagged text.

4.1.3.2 RQ2: When Uniqueness Is the Naming Rationale, does the Tagged Text Appear Directly in the Name or Is It Somehow Transformed?

To gain some additional insights in to how much extra work an automated tool might need, we further examined the tests in the Full and Partial categories to determine whether the information about what makes the test unique has the same form in the name or if it was transformed in some way. In order to get the information from the collected data, we performed a manual classification of all the tests in the full and partial categories. In this case were classified the tests as either Identical or Transformed.

- **Identical**: The tests in the Identical sub-category have this feature: each token from the test name that appears to come from the tagged text is identical to the token from the tagged text. For example, for `shouldIncludePort8080` from Scribejava, the tokens from the name are $\langle should,\ include,\ port8080\rangle$ from the name, and tokens from the tagged text are $\langle request,\ port8080\rangle$. The token `port8080` from the tokens of the name appears to come from the token `port8080` from the tokens of the tagged text, and they are identical to each other, so `shouldIncludePort8080` is included in the Identical category.

- **Transformed**: The tests in the Transformed sub-category have this feature: each token from the test name that appears to come from the tagged text

is transformed to the token from the tagged text, and it often indicates that both tokens from the name and tagged text have the same meaning but in different forms (i.e., abbreviation, plural and singular form, etc.). For example, for `shouldReturnUrlParam` from Scribejava, the tokens from the name are $\langle should, return, url, param \rangle$, and the tokens from the tagged text are $\langle get, parameter, null,$ a
The token "param" from the name appears to come from the token "parameter" from the tagged text, and it is a transformation of the token "parameter" from the tagged text by using the abbreviation of the word.

The results of this additional classification are also shown in Table 4.2. The Identical and Transformed sub-columns show the number of tests with their names in the identical and transformed categories, respectively. For example, the data from the third row shows the project Mockito has 4 tests that are fully named after the tokens from the tagged text. Of these 4 test names, 3 are under the Identical category, which indicates each of them has a shared subset of tokens between the tokens from its name and the tagged text. Each token in the shared subset is identical when comparing between the name and the tagged text (i.e., a physical intersection exists between the two sets of tokens from the name and the tagged text). The remaining test name is under the Transformed category. However, unlike for the other three, each token in the shared subset is transformed from its appearance in the tokens of the name to its corresponding appearance in the tokens of the tagged text (i.e., a conceptual intersection exists between the two sets of tokens from the name and the tagged text).

The third row also shows that Mockito has 27 tests that are partially named after the tokens from the tagged text. Of these 27 test names, 24 of them have the identical tokens between the name and the tagged text, and 3 of them has the same tokens between the name and the tagged text but the tokens of the name are transformed from the tokens of the tagged text. At the end, there are 9 tests in Mockito that are not named after the tokens from the tagged text at all. The data for each project slightly varies from each other, but the distribution of the identical and transformed

tokens are largely consistent. For example in Table 4.2, the number of identical tokens for each project is usually greater than the number of transformed tokens, and the majority of tokens from the name are identical to those from the tagged text (i.e., 22 out of 25 for full, and 261 out of 309 for partial). Therefore, RQ2 is answered with "yes" and "both" since both the identical and the transformed tokens were found in the comparison between the tokens from the name and the tagged text.

Judging from the results of both RQ1 and RQ2, the majority of the test names are indeed named after the tokens from the tagged text, and the tokens between the name and tagged text are mostly identical to each other, so we decided to proceed to build the automated tool.

4.2 Approach: Automated Identification of Unique Attributes

Figure 4.6 presents a high-level overview of our approach for automatically identifying the unique attributes of tests. As the figure shows, the approach takes a unit test and its siblings as input and identifies a set of unique attributes for the given test in two main steps. Step 1 is to extract attributes from the test and its siblings. The attributes are extracted using information matchers that correspond to the various selective codes identified as part of the empirical study in Section 4.1. Step 2 is to determine whether the extracted attributes are the portions of the test that make it unique from its siblings. If the attributes do make it unique, the approach proceeds to output the current attributes for the test. However, if the attributes are not unique, the approach returns to Step 1 and extracts a different set of attributes. The order in which Step 1 extracts attributes is based on the results of the empirical study; attributes that correspond to selective codes that are more likely to be what makes a test unique are tried before attributes that correspond to codes that are less likely to be unique. The loop between Step 1 and Step 2 proceeds until either a set of unique attributes are found or, if none of the attributes uniquely identifies the test, the approach terminates with an empty set of attributes. Additional details about each of the approach's steps are provided in the following subsections.

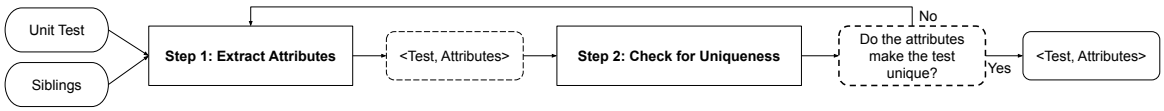
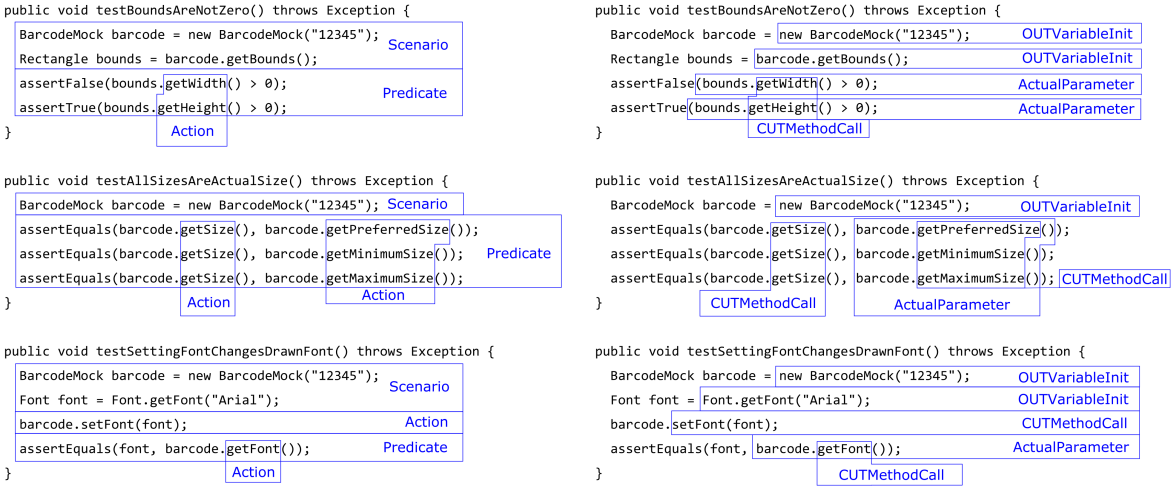


Figure 4.6: Overview of the automated approach.



(a) Primary Codes. (b) Secondary Codes.

Figure 4.7: Example outputs of information matchers.

4.2.1 Step 1: Extract Attributes

The goal of Step 1 is to extract attributes from each test that correspond to the information identified as part of the empirical study. Essentially, this step is attempting to automate the manual process of identifying relevant portions of tests that was used in the study. To do this, we built a set of information matchers, one for each selective code. These matchers extract attributes from the tests and produce a set of $\langle test,\ attributes \rangle$ pairs, one for each test given as input. Each $\langle test,\ attributes \rangle$ pair contains the test and its corresponding attributes, and each test is identified by its name in the following examples. This set of pairs are then used by the rest of the approach to generate a set of unique attributes.

Because the selective codes identify information at two levels (i.e., primary and

```
public void testDescendingIterator() {
  final RDequeRx<Integer> queue = redisson.getDeque("deque");
  sync(queue.addAll(Arrays.asList(1, 2, 3)));
  assertThat(toIterator(queue.descendingIterator())).toIterable().containsExactly(3, 2, 1);
}
```

Figure 4.8: Example of state-change methods.

secondary codes), there are two corresponding types of information matchers. Matchers for primary codes are primarily concerned with segmenting tests into their various parts (i.e., action, predicate, scenario). The division of the test into blocks is implemented primarily via elimination. First, the predicate matcher identifies calls to JUnit assertions and other predefined result-checking/verification methods. For example, in Fig. 4.3b, the call to `verify` is included as a predefined result-checking/verification method provided by the `Mockito` library. Such calls become the predicate block. Note that calls to methods embedded inside assertions (e.g., to calculate the actual parameter) or nested after assertions (e.g., to change state of the unit test) are not included as part of the predicate block. These calls are left for consideration by the matchers for the action and scenario.

Second, the action matcher identifies the remaining calls to CUT and non-CUT methods jointly with their corresponding arguments. Such calls become the action block. Note that those nested calls to non-CUT methods that reside after assertions are not included as part of the action block. These calls are left for consideration by the matchers for the scenario.

Finally, the scenario matcher identifies the OUT variable initializations, non-OUT variable initializations, control-flow variables, and calls to state-changing methods. The combination of these code elements becomes the scenario block. The scenario matcher first includes the calls to the state-changing methods of the test as part of the scenario block, which were previously excluded by the action matcher [119, 114]. For example in Fig. 4.8, for `testDescendingIterator` from `Redisson`, the call to the state-changing method is the first nested call after the assertion call: `toIterable`. Then it also selects all of the OUT variable initializations, non-OUT variable initializations

(i.e., differentiate from OUT by checking if it is used in the CUT method calls or assertions), and control-flow variables from the declaration and control-flow statements in the test.

As an example of how the high-level matchers work, consider the example shown in Fig. 4.7 which consists of three tests from Barbecue's `BarcodeTest` class. In the example, `testSettingFontChangesDrawnFont` is the target test and the other two are the siblings. The target test's predicate, action, and scenario are identified as follows: The matcher for the `Predicate` code marks all of the JUnit assertions in the test body as the predicate block (i.e., one statement circled in blue, tagged with `Predicate`). Next, the matcher for the `Action` code marks all of the CUT method calls in the test body as the action block (i.e., one full and one partial statements circled in blue, tagged with `Action`). Finally, the matcher for the `Scenario` code marks all of the OUT variable initializations in the test body as the scenario block (i.e., two statements circled in blue, tagged with `Scenario`).

Matchers for secondary codes are primarily concerned with identifying specific features within the blocks identified by the primary information matchers. To accomplish this, these secondary matchers select the corresponding code elements from the action, predicate, or scenario block. Because the secondary codes align with Java code elements (e.g., method call, object initialization, etc.), extracting them can be done with a straight-forward application of static analysis. Each matcher for the secondary codes is designed to search for and collect every code element that matches its specification. For example, the matcher for the `CUTMethodCall` code would extract all calls to methods declared by the class under test that are part of primary block under consideration.

As an example of how the information matchers for the secondary codes work, consider again the example shown in Fig. 4.7. Assume that the approach is extracting attributes for the ⟨`Action`, `CUTMethodCall`⟩ code. In this case, the matcher for the `CUTMethodCall` code searches in each test's action block and the set of ⟨*test*, *attributes*⟩ pairs that would be passed to Step 2 are:

- ⟨`testBoundsAreNotZero`, ⟨`getWidth`, `getHeight`⟩⟩
- ⟨`testAllSizesAreActualSize`, ⟨`getSize`, `getPreferredSize`, `getMinimumSize`, `getMaximumSize`⟩⟩
- ⟨`testSettingFontChangesDrawnFont`, ⟨`getFont`, `setFont`⟩⟩

Similarly, if the approach was extracting attributes for the ⟨`Scenario`, `OUTVariableInit`⟩ code, the matcher for the `OUTVariableInit` code searches in the test's scenario block and and the set of ⟨*test*, *attributes*⟩ pairs that would be passed to Step 2 are:

- ⟨`testBoundsAreNotZero`, ⟨`new BarcodeMock("12345")`, `barcode.getBounds()`⟩⟩
- ⟨`testAllSizesAreActualSize`, ⟨`new BarcodeMock("12345")`⟩⟩
- ⟨`testSettingFontChangesDrawnFont`, ⟨`new BarcodeMock("12345")`, `Font.getFont("Arial")`⟩⟩

4.2.2 Step 2: Check for Uniqueness

The goal of Step 2 is to identify whether the current set of attributes extracted by Step 1 make the target test unique among its siblings. This is accomplished using formal concept analysis (FCA) on the set of ⟨*test*, *attributes*⟩ pairs from Step 1 to build a concept lattice. The lattice is then analyzed to determine which, if any, of the target test's attributes, make it unique.

FCA is a data mining technique that is designed to facilitate the investigation of (implicit) relationships between a set of *objects* and a set of *attributes*. It has been successfully used in a variety of software engineering contexts (e.g., [111, 109]). In our approach, *objects* are the tests and *attributes* are the attributes extracted by Step 1.

As examples of a concept lattice, consider Fig. 4.9. As the figure shows, a concept lattice is a lattice that groups objects which share common attributes (i.e., a formal concept) and orders such groupings as a hierarchy (i.e., using subconcept/superconcept relations) of formal concepts [110]. A formal concept is a pair consisting of a subset of objects and a subset of attributes that is closed by Galois connection [101].

For example, if the formal concept is a pair like ⟨`objs`, `attrs`⟨, `objs` should consist of all objects that share the attributes in `attrs`, and `attrs` should consist of all attributes that are shared by the objects in `objs`. The topmost node in a lattice is the concept that contains all attributes, and the bottommost node is the concept that contains all objects [17]. In Fig. 4.9, each lattice was built for our running example from Fig. 4.7 using different sets of attributes.

In order to simplify the presentation, the lattices are shown in a reduced form, which means that each object and attribute only appears once, in the highest or lowest concept, rather than being duplicated in all super/sub-concepts. For example, in Fig. 4.9a the concept with no objects and attributes `getwidth` and `getheight`, is a super-concept of the two lower concepts which it connects to and a super-concept of the upper objects it connects to. This means that, beyond the attributes that are shown, this concept includes additional objects `testBoundsAreNotZero` and `testWidthAndHeightAreNotZero`.

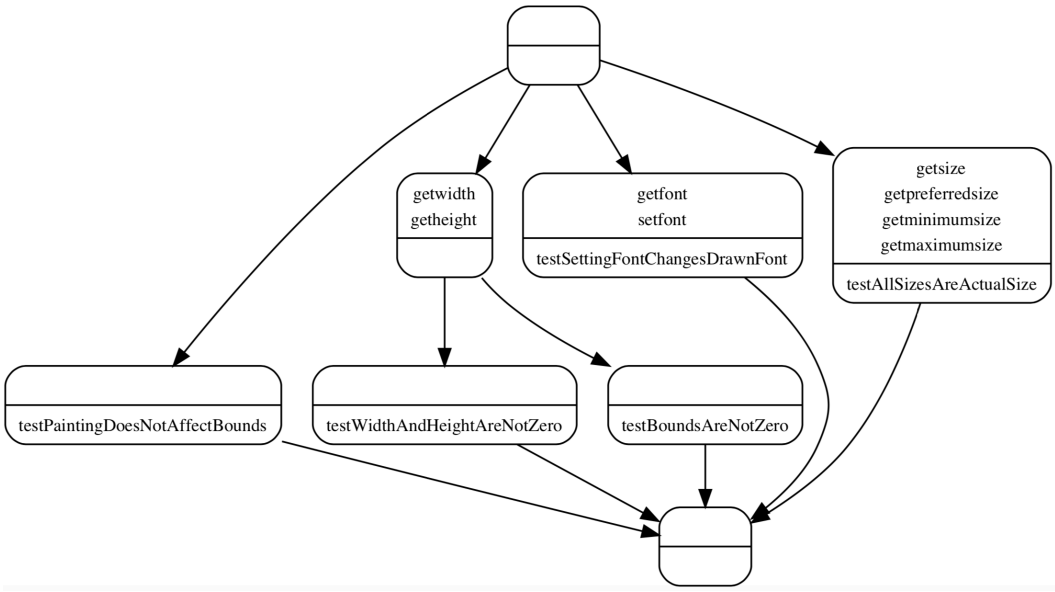
To check the uniqueness of a set of attributes, we implemented an algorithm to traverse the concept lattice. As input, the algorithm takes a concept lattice and a given test. The algorithm then performs a depth-first search of the concept lattice to attempt to locate a concept whose objects contains only the given test. If such a concept *can not be* be located, the approach returns to Step 1. Otherwise, if such concept *can be* located, the approach then extracts the subset of attributes that make the test unique.

The subset of attributes that make the test unique is extracted by looking at the reduced attributes of the identified concept. If the set of reduced attributes *is not empty*, it uniquely identifies the given test. For example, in Fig. 4.9a `testSettingFontChangesDrawnFont` is uniquely identified by the attributes: `getfont`, `setfont`. Otherwise, if the set of reduced attributes is empty, the algorithm returns to Step 1. For example, Fig. 4.9a shows that `testBoundsAreNotZero` can not be uniquely identified based on the CUTMethodCall code and another code needs to be considered. The algorithm moves on to the next code until it finds a set of reduced attributes that

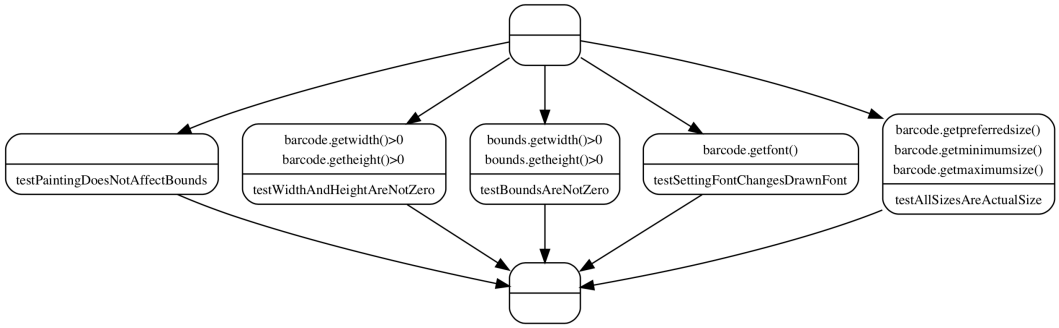
can uniquely identify the test. For example, while `testBoundsAreNotZero` can not be uniquely identified in Fig. 4.9a, Fig. 4.9b shows that it can be uniquely identified based on the ActualParameter code by the corresponding set of reduced attributes (`bounds.getwidth()>0`, `bounds.getheight()>0`). Finally, if all codes are tried without producing a valid set of reduced attributes, the algorithm outputs an empty set of attributes. For example, `testPaintingDoesNotAffectBounds` has an empty set of reduced attributes in all subfigures in Fig. 4.9.

As the last step, the attributes are processed to improve their legibility. First, white space and special symbols (e.g., "%" and "#") are removed. Second, if a predicate code was used to generate the attributes, some further manipulation is performed. Instances of "<", ">", "=", or numbers are replaced with corresponding English words like "GreaterThan" or "Zero" using a predefined lookup table. If the attribute starts with "set" or "get", it is prefixed by "When". If the attribute starts with "assert", it is prefixed by "If". Otherwise, it is prefixed by "Check". Third, if the attribute contains Java method calls, the stop words such as "set" or "get" are removed from them. Finally, the set of processed attributes is paired with the original test as the output of the approach.

As examples of unique attributes identified by the approach, consider Fig. 4.10. For the first test, the unique attribute (`offerFirst`) is identified for this test, which is fully consistent with the original name `testOfferFirst`. For the second test, the unique attributes (`CheckBoundsHeightGreaterThanZero,CheckBoundsWidthGreaterThanZero`) are identified for this test, which are partially consistent with the original name `testBoundsAreNotZero`.



(a) Concept lattice for attributes extracted using the Action-CUTMethodCall matcher.



(b) Concept lattice for attributes extracted using the Predicate-ActualParameter matcher.

Figure 4.9: Example concept lattices derived using FCA.

```
public void testOfferFirst() {
    RDequeRx<Integer> queue = redisson.getDeque("deque");
    sync(queue.offerFirst(1));
    sync(queue.offerFirst(2));
    sync(queue.offerFirst(3));
    assertThat(sync(queue)).containsExactly(3, 2, 1);
}

Unique Attributes: [offerFirst]

public void testBoundsAreNotZero() throws Exception {
    BarcodeMock barcode = new BarcodeMock("12345");
    Rectangle bounds = barcode.getBounds();
    assertFalse(bounds.getWidth() > 0);
    assertTrue(bounds.getHeight() > 0);
}

Unique Attributes: [CheckBoundsHeightGreaterThanZero,CheckBoundsWidthGreaterThanZero]
```

Figure 4.10: Example output showing the unique attributes for two tests.

4.3 Evaluation: Automated Identification of Unique Attributes

The overall goal of empirical evaluation is to determine if our approach can match human judgment and to estimate the amount of effort needed to transform the extracted attributes into descriptive test names. More specifically, we considered three research questions:

(1) **RQ1—Feasibility**. Can the approach automate our guidelines for extracting unique attributes?

(2) **RQ2—Consistency**. Do the attributes identified by the approach agree with human judgement?

(3) **RQ3—Effort**. How much effort may be needed to transform the extracted attributes into descriptive names?

In the context of future work, these are the most important evaluation metrics. If the approach can not accurately identify what makes a test unique or if it does not agree with human judgement, it can not be used for generating descriptive names. Or if our extracted attributes requires too much effort to be transformed into descriptive test names, it might not be suitable to be the foundation of future name generation approaches.

In order to conduct the evaluation, we implemented the approach as an IntelliJ IDEA Plugin [62]. IntelliJ IDEA is a full-featured IDE that can import projects from many different build systems (e.g., Maven and Gradle). This gives us more flexibility in choosing applications when building our set of experimental subjects. To use the plugin, developers can click on a menu item that analyzes all tests in the current project and it could easily be extended to support other interaction mechanisms (e.g., to run only for a specific test class).

The plugin was used to gather the necessary data for performing the evaluation. When running on a MacBook Pro (2.4GHz Intel i5 processor and 8GB RAM) with MacOS Mojave, Kotlin version 1.3.10, and Java version 13.0.2, the plugin analyzed

Table 4.3: Additional considered applications.

Project	Version	LoC	# Tests
Sentinel	10c92e6	79,385	488
Jedis	d7aba13	36,582	684
Jfreechart	4e0a53e	133,540	2,176
Redisson	6feb33c	150,703	2,036
Spark	7551a7d	11,956	310
Webmagic	96ebe60	15,774	98

all 31,251 tests from the 17 projects considered in the evaluation of RQ1 in about 78 minutes (i.e., an average of 0.15 seconds per test). We believe that this level of performance is reasonable and is fast enough to support using the tool as part of future name generation approaches.

The remainder of this section describes methodology and results for each research question in more detail.

4.3.1 Considered Subjects

As the subjects for RQ1, we started with the 440 labeled tests that we used in the empirical study. Because of the significant amount of human work in manually identifying what makes a test unique, it makes sense to reuse them. However, because of the threat that the approach may be over-fit to these subjects, we decided to augment them. To do this we first chose 6 additional projects from GitHub. The additional projects, shown in Table 4.3, were selected using the same rational that was used for selecting the 11 applications from the study. Then, we again sampled the set of tests in order to manage the amount of manual effort. In this case, we selected 480 tests, 80 from each application. Finally, using the same procedure as for the empirical study, each of the 480 tests was manually analyzed to identify what makes it unique among its siblings. As a result, we have 920 tests, labeled with what makes them unique, that serve as the subjects for this part of the evaluation. A copy of this data set is publicly available, and a Readme file is also provided [66].

testFindRangeBounds

```
@Test
public void testFindRangeBounds() {
     AbstractXYItemRenderer renderer = new StandardXYItemRenderer();
     // check that a null dataset returns null bounds
     assertTrue(renderer.findRangeBounds(null) == null);
}
```

Figure 4.11: Example of a participant-labeled test.

In order to address a potential source of bias associated with using data labeled by the authors, we recruited three participants to create an additional data set for RQ2 and RQ3. The participants are PhD students at the University of Delaware who are unaffiliated with this work. In addition, they each have at least three years of experience with Java and JUnit.

The new data set consists of 45 tests, 5 randomly selected from 9 projects, the 6 additional projects selected as part of RQ1 and Picasso, Fastjson, and Moshi. Each participant was asked to consider each test and its siblings and then to annotate the parts of the test that they believe make it unique. Note that participants were not informed about the research nor the guidelines developed as part of the empirical study before performing the task. An example of the full instructions given to each participant are publicly available [66]. In total, a labeling session took each participant around 4 hours to complete. An example of the results of the labeling process is shown in Fig. 4.11. In the figure, the part of `testFindRangeBounds` that the participant believes makes it unique, the method call `findRangeBounds` inside the last assertion, is shown with a green highlight. A copy of all collected data is publicly available, and a Readme file is also provided [66].

4.3.2 RQ1: Feasibility

The goal of RQ1 is to evaluate whether the approach can automate our guidelines for extracting unique attributes. This serves as a feasibility check for the implementation.

To answer the question about whether the approach can automate our guidelines

Table 4.4: Data showing the consistency between author-provided annotations and the output of the approach.

	Levels of consistency (%)			
Project	**Equivalent**	**Approach+**	**Approach−**	**Mismatch**
Barbecue	25.0	10.0	50.0	15.0
Moshi	35.0	37.5	12.5	15.0
Mockito	30.0	35.0	25.0	10.0
Guava	32.5	37.5	22.5	7.5
Guice	10.0	45.0	32.5	12.5
ExoPlayer	45.0	32.5	12.5	10.0
Scribejava	32.5	27.5	37.5	2.5
Socket.io-client	35.0	40.0	15.0	10.0
Fastjson	35.0	32.5	30.0	2.5
Picasso	47.5	12.5	25.0	15.0
Javapoet	12.5	40.0	30.0	17.5
Sentinel	25.0	58.8	12.5	3.7
Jedis	37.5	30.0	32.5	0.0
Jfreechart	42.5	36.3	21.2	0.0
Redisson	40.0	22.5	33.7	3.8
Spark	31.2	41.3	27.5	0.0
Webmagic	7.5	33.8	55.0	3.7
Overall	30.8	34.6	28.5	6.1

for extracting unique attributes or not, it is necessary to establish a procedure for comparing the manually identified attributes with the automatically identified attributes. To do this, we employed a manual review and comparison process that is similar to one used in the empirical study (see: Section 4.1). Together, the authors compared the manually identified attributes with the automatically identified attributes of each test and classified the tests into one of the categories below. In total, performing the 920 comparisons took around 20 hours.

(1) Equivalent: A test is included in the Equivalent category when (1) the two sets of attributes contain the same number of elements, and (2) each element in one

set expresses the same information as an element in the other set. For example, `testOfferFirst` from Redisson is included in the Equivalent category because its set of automatically identified attributes (`OfferFirst`) is the same as the manually identified attributes (`OfferFirst`). Similarly, `testSerialization` from Guava is also included in the Equivalent category because its set of automatically identified attributes (`reserialize`) and its set of manually identified attributes (`reserialized`) represent the same information (i.e., "reserialize" and "reserialized" differ only in their tenses).

(2) Approach Plus (+): A test is included in the Approach Plus category when (1) the set of automatically identified attributes contains more elements than the set of manually identified attributes, and (2) each element in the set of manually identified attributes expresses the same information as an element in the set of automatically identified attributes. For example, `clientIdReconnect` from Jedis is included in the Approach Plus category because its set of automatically identified attributes (`disconnect, connect`) is larger than its set of manually identified attributes (`disconnect`) and the element in the manually identified set (`disconnect`) is also present in the automatically generated set.

(3) Approach Minus (−): A test is included in the Approach Minus category when (1) the set of automatically identified attributes contains fewer elements than the set of manually identified attributes, and (2) each element in the set of automatically identified attributes expresses the same information as an element in the set of manually identified attributes. For example, `testExitLastEntryWithDefaultContext` from Sentinel is included in the Approach Minus category because its set of automatically identified attributes (`getFakeDefaultContext, runOnContext`) contains fewer elements than its set of manually identified attributes (`getFakeDefaultContext, runOnContext, defaultContext.getCurEntry`) and each element in the set of manually identified attributes is also in the set of automatically identified attributes.

(4) Mismatch: A test is included in the Mismatch category when it does not meet the definitions of any of the above categories. For example, `flattenTopLevel` from Moshi is included in the Mismatch category because its set of automatically identified attributes (`hasMessage`) has nothing in common with the set of manually identified attributes (`"Nesting problem."`).

A result in the first three categories indicates cases where the approach is able to follow the guidelines that we provided for extracting unique attributes from JUnit tests. For tests in the Equivalent category, our approach identifies the same unique attributes as the guidelines describes, which could be directly applied to future name generation approaches. For tests in the Approach Plus category, our approach identifies the same unique attributes as the guidelines describes but includes some additional information. And for tests in the Approach Minus category, our approach identifies some of the same unique attributes the guidelines describes. Only in the case of the Mismatch category does the approach fail to provide useful information.

Table 4.4 presents the results of the classification process. In the table, the first column shows the name of each project and the following four columns show the percentage of tests in the Equivalent, Approach Plus, Approach Minus, and Mismatch categories, respectively. The first eleven rows show the data for the 11 original projects, the following six rows show the data for the 6 additional projects, and the final row shows the distribution across all projects. For example, the twelfth row shows that for the considered tests from Sentinel, the Equivalent, Approach Plus, Approach Minus, and Mismatch rates are 25.0 %, ≈58.8 %, ≈12.5 %, and ≈3.7 %, respectively.

Overall, we believe the performance of the approach is positive judged by our authors. As the last row of the table shows, the average Equivalent, Approach Plus, and Approach Minus rates are ≈30.8 %, ≈34.6 %, and ≈28.5 %, respectively, while the average Mismatch rate is only ≈6.1 %. This means that in ≈93.9 % of cases, the approach can automate our guidelines for extracting unique attributes.

While the overall performance is strong, the data shows that our approach performed noticeably better on some projects than others. For example, the approach has a 0.0 % Mismatch rate on Jedis and Jfreechart, while it has a ≥15.0 % Mismatch rate on Picasso and Javapoet. To understand the causes of the variation and to potentially identify avenues for improving the approach, we further investigated the cases in the Approach Plus, Approach Minus, and Mismatch categories.

First, for the subjects from the Mismatch category, we found that there are many cases (i.e., ≥60.0 %) where the human-identified attributes were extracted using a lower-ranked code than the code used by the approach. For example, for `should-HandleSchemeInsensitiveCase` from Picasso, the approach used a higher-ranked code `Action-OtherMethodCall`, but the human judgment corresponds to a lower-ranked code `Predicate-ActualParameter`. In these cases, if the approach were to use the lower-ranked code, it would identify the same attributes as the human rater. This suggests that, while our current ordering of the codes is effective for many projects, it is not universally suitable. In future work, we could investigate ways of customizing the order of code for individual projects.

Second, for the subjects from the Approach Minus category, we found that the attributes that were missing from the set of automatically identified attributes could always be found by using an additional code or codes. For example, for `null-BitmapOptionsIfNoResizingOrPurgeable` from Picasso, the manually identified attributes are (`noResize, isNull()`) and the automatically identified attributes are (`noResize`). The missing attributes (`isNull()`) could be identified by using the code `Predicate-ExpectedParameter` which is lower-ranked than the code `Action-Other-MethodCallArgument` which was used to extract the automatically identified attributes. This suggests that extending the approach to consider multiple codes may result in identifying attributes that more difference indicates it might be useful to extend the approach to be capable of applying multiple codes when extracting unique attributes.

Last, for the subjects from the Approach Plus category, we found that the additional attributes identified by the approach are unnecessary to uniquely identify

Table 4.5: Precision and recall for different projects.

	Percentage (%)		
Project	**Precision**	**Recall**	**F1**
Picasso	100.0	100.0	100.0
Fastjson	100.0	100.0	100.0
Moshi	100.0	100.0	100.0
Redisson	100.0	100.0	100.0
Spark	100.0	100.0	100.0
Webmagic	88.9	100.0	94.1
Jedis	100.0	100.0	100.0
Sentinel	50.0	100.0	66.7
Jfreechart	100.0	100.0	100.0

a given test among its siblings. For example, for `testSlowRequestMode` from Sentinel, the manually identified attributes are (`current, nextInt`), and the automatically identified attributes are (`setSlowRatioThreshold, entryAndSleepFor, nextInt, current`). While the automatically identified attributes do uniquely identify the test, the attributes (`setSlowRatioThreshold, entryAndSleepFor`) are unnecessary; (`current, nextInt`) are sufficient. These additional attributes are included because the approach extracts all attributes in a given category. In this case, the code used by the approach is `Action-OtherMethodCall` and the attributes are all of the non-CUT method calls in the test. In future work, we could potentially address this issue by modifying the approach to check whether subsets of attributes are sufficient to uniquely identify test (i.e., if all attributes uniquely identify a test, determine the smallest subset that is sufficient).

4.3.3 RQ2: Consistency

The goal of RQ2 is to evaluate whether the unique attributes identified by the approach agree with human judgment. If the attributes identified by the approach are significantly different from what humans believe are the unique aspects of a test it is unlikely that the approach will be useful for generating descriptive names.

Inter-rater reliability measures are a standard way of assessing levels of agreement. In our situation, Fleiss' Kappa makes sense as it supports multiple raters (i.e., our participants). However, Fleiss' Kappa is designed to operate on categorical data so it is necessary to transform both the attributes identified by the approach and the annotations provided by the participants to a unified form.

To transform our data, we created a set of boolean questions, one for each line in each test case, where each question asks whether the approach or the participants consider the line to contain a unique aspect of the test. A participant's answer to a question is true if they highlighted part of the corresponding line and false otherwise. Similarly, the approach's answer to a question is true if an attribute it identifies is taken from the corresponding line and false otherwise. We chose to work at the line-level because we found that alternatives such as considering tokens or characters were too low-level and were unnecessarily sensitive to inconsequential differences (e.g., was a terminating semicolon highlighted or not?).

After transforming the data, we first investigated the level of agreement among the raters and calculated a kappa score of 0.27. Under the standard interpretation metric, this is within the "fair" range of 0.21 to 0.40. This suggests that the participants often have different opinions about what makes a test unique among its siblings and that **there is not a single "correct" answer** to which we can compare our approach. Because there is not a single correct answer, we decided to further investigate this question in two ways (1) looking at the agreement between the approach and each participant individually, and (2) calculating precision and recall when comparing the approach against any participant.

When investigating the level of agreement between the approach and each participant, we found kappa scores of 0.68, when comparing against Participant 1, 0.28, when comparing against Participant 2, and 0.23, when comparing against Participant 3. Overall, we believe that these results are positive. The level of agreement between the approach and Participant 1 is "substantial" and the levels of agreement between the approach and Participants 2 and 3 are "fair" and similar to the level of agreement

among all participants. This suggests that the attributes identified by the approach can pass for human judgement.

To compare to any participant, we calculated precision and recall where true positive (TP), false positive (FP), true negative (TP), and false negative (FN) are defined as follows. A true positive occurs when the approach's answer to a question is true and *at least one* of the participants' answers is true. A false positive occurs when the approach's answer to a question is true and *all* of the participants' answers are false. A true negative occurs when the approach's answer to a question is false and *at least one* of the participant's answers is false. A false negative occurs when the approach's answer to a question is false and *all* of the participants' answers are true.

Under these definitions, the average precision and recall for the 5 selected tests from each project are shown in Table 4.5. In the table, the first column, *Project*, shows the name of each project, and the second and third columns, *Precision* and *Recall*, show the average precision and recall, respectively, for each the corresponding project. For example, the sixth row shows that the average precision and recall for the questions from the tests for `Webmagic` are ≈88.9 % and 100.0 %, and the average F1-score for the questions from the tests is ≈94.1 %.

As the table shows, the average precision and recall across most projects is high. This indicates that the attributes identified by the approach nearly always match with the judgement of at least one participant. One exception to this trend is the lower precision for `Sentinel`. In this case, we found the specific application domain of `Sentinel` likely causes the difference. As we figured out that `Sentinel` is not an ordinary Java project like others in our evaluated project but a Java-based throttling-control framework [5]. Because of its domain and our conservative approach, their testing is more likely to be designed for verifying software performance and flow of network traffic, not normal JUnit tests. For example in `AbstractSofaRpcFilterTest` of `Sentinel`, because this class is intended to test a service provider filter that requires many setups and subtle comparisons, every test (e.g., test service performance) in this class is hard to perform any static analysis on. Therefore, our raters are more

Test body

```
{
    Picasso picasso = mock(Picasso.class);
    ImageView target = mockImageViewTarget();
    Callback callback = mockCallback();
    ImageViewAction request = new ImageViewAction(picasso, target, null, false, false, 0, null, URI_KEY_1, callback);
    request.cancel();
    assertThat(request.callback).isNull();
}
```

Question 1:

Relevant words/phrases for naming the test:

```
cancel
isNull
```

- Are there words/phrases that should be added?

  `ImageViewAction`

- Are there words/phrases that should be removed?

- Use the words/phrases to name the test

  `ImageViewActionCancelIsNullTest`

Figure 4.12: Example of a participant response for constructing descriptive name.

likely to identify those subtle differences (e.g., order of service parameters, number of specific assertion calls, etc) as experienced developers and choose different elements than our approach for what makes the test unique. Surprisingly, in Table 4.4 from RQ1, comparing to other projects, we can also see a significant increase of tests categorized as Approach Plus (i.e., our approach picked up more attributes than humans) for `Sentinel` at the twelfth row. This observation also supported our assumption about why `Sentinel` had a lower precision than others. Despite this outliner, these results further emphasize our belief that the approach is capable of extracting information about what makes a test unique that agrees with human judgment.

Additionally, we also looked into what kind of statements developers often agree on as being what makes the test unique. For example, some developers might prefer to select the uniqueness of test from assertions, and others might not. Overall, from our three participants, the count of assertions that get selected as the uniqueness of test is about equal to the count of non-assertion statements. We further investigate on this topic with more subjects as future work.

4.3.4 RQ3: Effort

The goal of RQ3 is to investigate how much effort may be needed to transform the extracted attributes into descriptive test names. To help place the results for our approach in context, we also considered two alternative approaches. The first alternative is Code2vec [10]. We chose Code2vec because it is a state-of-the-art machine translation approach for generating method names that significantly outperforms alternative approaches (i.e., [8, 57, 9]). The second alternative is the original, developer provided name for the test.

Because our approach produces sets of attributes, it was necessary to convert the original name and the name generated by Code2vec into a comparable form. For the original names, we accomplished this by using an identifier splitter to break the name into tokens. These tokens served as the attributes for the original name. For Code2vec, we were able to slightly modify the implementation to output the predicted set of words before they were joined to form a name. These words served as the attributes for Code2vec.

To compare the results of the approaches, we asked human participants to answer a series of questions about the attributes identified by each approach. The participants we used for this part of the evaluation are the same three students that we used for RQ2. To avoid potential biases related to the participants having already seen the tests, we created a new data set for RQ3. Like we did for RQ2, we randomly selected 5 tests from 9 projects for a total of 45 tests. But in this case, we made sure not to select a test that was used in the investigation of RQ2. We then used each approach to generate a set of attributes for each test. Each of the 135 sets of attributes was then used to generate a series of questions. Figure 4.12 shows an example of the questions for each attribute. First, we showed each participant the test body and the set of attributes. Then we asked, if they were to create a name for the test based on the attributes: (1) if any words or phrases should be added, (2) if any words of phrases should be removed, and (3) what name they would chose.

Figure 4.13: Data showing the count of modifications for each approach and for each project.

Each participant answered each set of questions and took approximately 8 hours in total to answer all questions. In the example shown in Fig. 4.12, participant believes that the phrase "ImageViewAction" should be added, nothing should be removed, and that an appropriate name for the test is "ImageViewActionCancelIsNullTest".

As a post-processing step, we slightly edited the responses to remove edits strictly related to formatting. For example, additions of the word "test" where the intent was to use it to prefix the test name were removed. This process was done in consultation with the participants to ensure that their original intentions were retained.

To quantify the amount of manual effort that may be needed to convert a set of attributes generated by an approach into a descriptive name, we first looked at the number of modifications that the participants believe are necessary.

Figure 4.13 presents a series of scatter plots that show the total number of modifications (i.e, additions plus removals) that each participant would make to attributes identified by each approach for each test. The plot is faceted horizontally by project (i.e., each column shows the results for a project) and vertically by participant (i.e., each row shows the results for a participant). Within each plot, each tick on the x-axis represents a test from the corresponding project, the y-axis shows the number of

modifications, and the color of a point indicates which approach was used to generate the attributes. Note that a small amount of horizontal jitter was applied to each point to make situations where multiple approaches require the same number of modification more obvious. For example, the upper-left most plot shows that Participant 1 believes that, for the first test selected from Fastjson (left-most tick on the x-axis), the attributes identified by both Code2vec and the original name require 2 modifications and the attributes identified by our approach require 0 modifications.

From this data, we looked at each combination of test (45) and participant (3) and calculated: (1) how often each approach requires 0 modifications: ≈4 % (6 out of 135) for Code2vec, ≈24 % (32 out of 135) for the original name, and ≈46 % (62 out of 135) for our approach,, (2) how often each approach requires the fewest modifications: ≈25 % (34 out of 135) for Code2vec, ≈44 % (60 out of 135) for the original name, and ≈67 % (90 out of 135) for our approach, and (3) how often each approach requires the most modifications: ≈47 % (64 out of 135) for Code2vec, ≈51 % (69 out of 135) for the original name, and ≈31 % (42 out of 135) for our approach.

We also investigated, again for each combination of test and participant, how much worse each approach performs compared to the best approach. To do this, we first identified the fewest number of modifications required across the approaches. For example, if, for Test 1, Participant 1 indicates 6 modifications for Code2vec, 4 modifications for the original name, and 1 modification for our approach, the fewest number of modifications is 0. We then subtracted this number from the number of modifications required by each approach to compute the increase in the number of modifications required by each approach over the best. Using the example from above, the increase is 5 for Code2vec ($6 - 1$, 3 for the original name ($5 - 1$, and 0 for our approach ($1 - 1$.

Table 4.6 shows the mean of the increase in the number of modifications across each project and across the entire data set. In the table, the first column, *Project*, shows the name of each project, and the next three columns, *Code2vec*, *Original*, and *Ours*, show, for the corresponding approach, the mean increase in the number of

Table 4.6: Mean increase in the number of modifications over the fewest number of modifications.

Project	Code2vec	Original	Ours
Fastjson	1.00	0.47	0.73
Jedis	2.40	1.07	0.87
JFreeChart	1.73	1.40	0.13
Moshi	1.27	1.47	1.27
Picasso	1.13	2.80	0.67
Redisson	2.27	1.20	0.87
Sentinel	1.93	2.07	1.20
Spark	3.00	4.20	2.07
Webmagic	2.33	1.33	1.00
Overall	1.90	1.78	0.98

modifications over the best approach. The final row in the table shows the mean of the increase for each approach over the entire data set. For example, the final row shows that the attributes identified by Code2vec, the original name, and our approach required, on average, 1.9, 1.78, and 0.98 more modifications than the approach that required the fewest modifications.

Across all four these metrics, our approach performs better than both Code2vec and the original name. It requires the most modifications the least often, the fewest modifications the most often, and, more importantly, 0 modifications the most often. In addition, its mean increase in the number of modifications is lower than both Code2vec and the original name overall and for 7 out of 9 projects. The only exceptions are for Fastjson where the original name has a lower mean increase (0.47 compared to 0.73) and for Moshi where Code2vec has the same mean increase (1.27). This means that, among the considered approaches, the our approach is the most likely to identify the attributes that require the least modification and are therefore the most useful for generating descriptive names.

In addition to looking at the data in terms of modifications, we also considered the ratio between the number of additions and the number of removals. This is a potentially interesting metric as the types of modifications may not have the same

Table 4.7: Mean ration of additions to removals.

Project	Code2vec	Original	Ours
Fastjson	0.58	0.59	0.52
Jedis	0.49	0.44	0.50
JFreeChart	0.61	0.63	0.83
Moshi	0.50	0.31	0.52
Picasso	0.74	0.20	0.71
Redisson	0.42	0.40	0.66
Sentinel	0.54	0.32	0.37
Spark	0.62	0.31	0.32
Webmagic	0.55	0.60	0.37
Overall	0.56	0.42	0.53

cost. For example, additions may be more expensive than removals as additions require additional comprehension work to create something new whereas removals do not. To calculate the ratio of additions to removals, we first filtered instances where a participant indicated that the attributes identified by an approach required 0 modifications. This left us with 305 of the original 405 data points. We then calculated the percentage of modifications that are additions (e.g., 3 additions and 2 removals results in a 60 % addition ratio.). Table 4.7 shows the results of this calculation in the same format as Table 4.6. For example, the first row shows that, for Fastjson, the mean ratio of additions to removals for Code2vec, the original name, and our approach is 0.58, 0.59, and 0.52, respectively.

Across the entire data set, the ratio of additions to removals is close to 50 % with Code2vec and our approach tending to have more additions than removals and the original name tending to have more removals than additions. Within a project though, the ratio is sometimes more lopsided. For example, for JfreeChart the ratio of additions to removals for our approach is 0.83 while for Spark it is 0.32. We believe that these results indicate that none of the approaches is significantly biased towards requiring more additions or removals. This means that, overall, the total number of modifications is the more important factor to consider when comparing the suitability

of the approach for identifying attributes that will be used to generate descriptive names. Therefore, our approach is more likely to be useful in future name generation approaches since it requires less effort to be transformed into descriptive test names. Based on this observation, it is possible for us to develop a uniqueness-based approach to generate descriptive JUnit test names.

4.4 Empirical Study: How are Unique Aspects Transformed to a Name?

Based on the identified attributes of test in Section 4.2 and a set of attribute transformations, we developed a uniqueness-based approach that can generate descriptive JUnit test names with developers' approval. In order to develop the uniqueness-based name generation approach, we conducted an empirical study to learn how the unique aspects of a test are transformed into a name.

As the basis for the uniqueness-based name generation approach, we are using our recent work on automatically identifying the unique aspects of a test [115]. That work was motivated by a large-scale empirical study of the tests from 11 projects that demonstrates that the unique aspects of a test are often the basis for its name. Based on the results from the empirical study, we designed a novel, automated approach that can extract the attributes of a test that make it unique among its siblings. At a high-level, the approach uses a combination of static program analysis—to extract a variety of candidate attributes from a test suite—and formal concept analysis—to identify which combination of attributes is unique to the test under consideration.

For example, Fig. 4.14 shows a test and the output of the uniqueness-identification approach. The identified attributes of the test are: `putAsync` and `getAsync` (both are calls to methods defined by the class under test). Those attributes are unique because no sibling (i.e., tests in the same class) contains both method calls. While the output of the approach agrees with human judgement about what features of a test make it unique, they can not be directly used as a test name.

In order to conform with expected naming conventions, the unique attributes of a test must be transformed and merged in various ways. For the example from Fig. 4.14,

```
public void ***() throws Exception {
      RedisProcess runner = new RedisRunner()
              .nosave()
              .randomDir()
              .port(6311)
              .run();
      URL configUrl = getClass().getResource("redisson-jcache.yaml");
      Config cfg = Config.fromYAML(configUrl);
      Configuration<String, String> config = RedissonConfiguration.fromConfig(cfg);
      Cache<String, String> cache = Caching.getCachingProvider().getCacheManager()
              .createCache("test", config);
      CacheAsync<String, String> async = cache.unwrap(CacheAsync.class);
      async.putAsync("1", "2").get();
      assertThat(async.getAsync("1").get()).isEqualTo("2");
      cache.close();
      runner.stop();
}

Identified Attributes: putAsync, getAsync

Attributes Additions: Cache

Abstract of Attributes: (-put, -get) Async

Append prefix: Test

Join as Name: TestCacheAsync
```

Figure 4.14: From attributes to a descriptive name.

the test name `TestCacheAsync` shows the result of this process. This name was created, based on the unique attributes, by a human participant involved the in evaluation of the uniqueness-identification approach. In this case, it appears that the participant used three transformations to construct the name from the unique attributes. First, the existing attributes are merged into a single attribute, `Async`, by removing the leading `put` and `get`. Second, an additional attribute—`Cache`, a variable name of the class under test—is added to the set of attributes. Third, the attributes are ordered, in this case, based on execution order. Finally, each word in the attributes are capitalized and the resulting attributes are joined together, in order, with a leading `Test`. In order to create a successful approach for generating test names based on unique attributes, it is necessary to understand the types of modifications and transformations that would be performed by a human in more details. The next section describes our processing for investigating this question.

Table 4.8: Considered projects.

Project	Version	LoC	# Tests
Sentinel	10c92e6	79,385	488
Jedis	d7aba13	36,582	684
Jfreechart	4e0a53e	133,540	2,176
Redisson	6feb33c	150,703	2,036
Spark	7551a7d	11,956	310
Webmagic	96ebe60	15,774	98
Picasso	a087d26	11,006	229
Fastjson	e05f1f9	195,511	4,950
Moshi	dbed99d	22,168	716

4.4.1 Considered Subjects

In order to understand the types of transformations that are often made to create a descriptive test name, we used data collected from the evaluation of the uniqueness-identification approach [115]. This evaluation asked 3 participants about the performance of the uniqueness-identification approach when applied to 45 tests randomly selected from 9 top-ranked GitHub Projects (i.e., 5 tests from each project). The considered projects can be found in Table 4.8. As part of the evaluation, the participants were asked to consider a test body and the unique attributes that were identified by the approach. Then they were asked, if they were to create a name for the test based on the attributes: (1) if any words or phrases should be added, (2) if any words of phrases should be removed, and (3) what name they would chose.

4.4.2 Selective Coding Process

At a high-level, the participants' answers to the survey questions give a rough structure to how the unique attributes need to be transformed to create a name: Addition, Removal, and Modification.

Figure 4.15 shows the distribution of how the unique attributes were transformed by each participant. In the figure, the y-axis shows each type of transformation made by each participant (i.e., change), and the x-axis shows how many times each participant

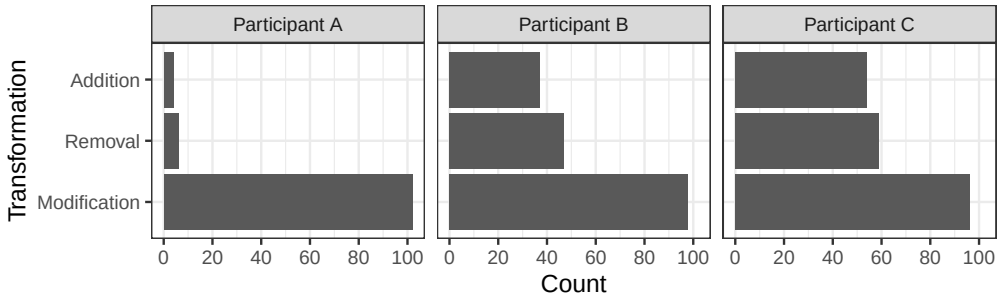


Figure 4.15: Proportion of change by each participant.

made each type of transformation (i.e., count). Note that multiple additions, removals, or modifications, may be done for each test. For example, Participant A made 4 additions, 6 removals, and 102 modifications while Participant B made 37 additions, 47 removals, and 98 modifications. We can make two observations based on the data shown in Fig. 4.15. First, for each participant, the number of additions and removals is often less than the number of modifications. This observation indicates that additions and removals are not always needed when transforming the unique attributes to a name but modifications are often needed. Second, the count of modification for each participant is often significantly greater than the number of tests—45. This observation indicates, even if there is no additions and removals made to the unique attributes, modifications are always needed to transform the unique attributes to a name.

To further understand each of the transformation type, we used an open, axial, selective coding process. Moreover, after we completed the coding process, we consulted with the original participants as a sanity check. During an individual, online meeting, each participant was shown their answers and the results of the coding process. In all cases, the participants' felt that the outcome of the coding process accurately captured their intent and the rationales behind their choices. In addition, the participants expressed interest in eventually using an implementation of the name generation approach as they believe that it would help them generate acceptable names more quickly.

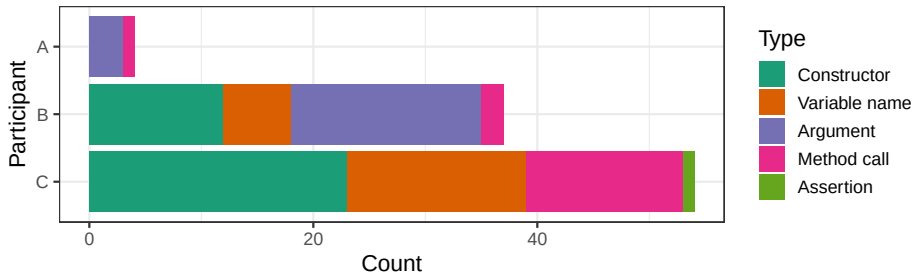


Figure 4.16: Proportion of types of attribute additions.

4.4.3 Result and Discussion

In this section, we discuss the result of the selective coding process through some descriptive data presented in Figs. 4.16, 4.17 and 4.18. Overall, we coded three types of change: Addition, Removal, and Modification. Each of them represents a specific type of change that developers often made to transform the unique aspects of a test into a name.

4.4.3.1 Addition

Addition occurs when a participant adds additional words or phrases to the set of attributes identified by the approach. As a result of the coding process, we identified the following types of addition:

Constructor: the name of the class under test or a constructor used in the assertions from the test body. For example, in `runWithIoExceptionDispatchRetry` from `Picasso`, `BitmapHunter` is the name of class under test and is added to the set of attributes.

Variable name: the name of a variable that contains a instance of the class under test or is used in an assertions. For example in `bitmapConfig` from `Picasso`, `data` is the name of a variable that contains a instance of the class under test that is added.

Argument: an argument that is passed to a focal method call [40]. For example, in `testConcurrentGetMachines` from `Sentinel`, `concurrent` is a method argument that is extracted from the focal method calls and added to the set of attributes.

Method call: a call to the method that is declared by the class under test. For example, in `nameString` from `Jedis`, `clientSetname` is a call to a method that is declared by the class under test. Often, the methods that are called most frequently are added.

Assertion: a call to an assertion used in the test body. For example, in `withoutFallbackValue` from `Moshi`, `fail` is added to the set of attributes.

Figure 4.16 shows the number of times each type of addition occurred as a stacked barchart. In the figure, the y-axis shows each participant, the colors show the type of addition, and the x-axis shows how many times each participant made each type of addition. For example, from each type of addition's perspective, Participant A makes two types of attribute addition: 3 times of Parameter additions, and 1 times of Method call additions.

Based on the data from Fig. 4.16, it appears that what to add is a personal choice. For example, while Participant A often preferred to add Parameters and Method calls, Participant B often preferred to add Constructors and Variable names. This suggests that a fully automated name generation approach is difficult and a cooperative approach that works with developers may be preferred.

4.4.3.2 Removal

Removal occurs when a participant removes existing words or phrases from the set of attributes identified by the approach. As a result of the coding process, we identified the following types of removal. Note that, with the exception of Assertion, the types of removal are essentially the inverse of the types of addition.

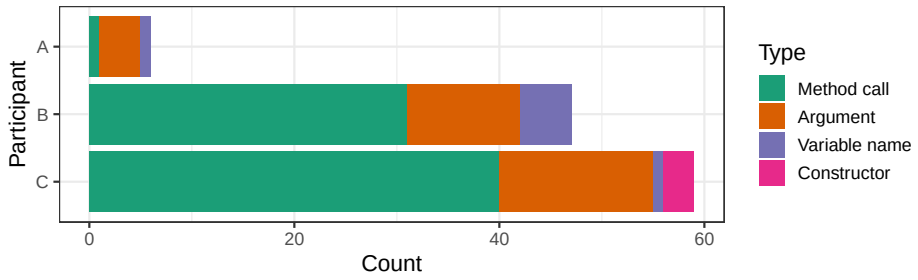


Figure 4.17: Proportion of type of attribute removals.

Constructor: a constructor that is not declared by the class under test. For example, in `testInnerEntry` from `Fastjson`, `InnerEntry()` is a constructor that is not the name of the class under test that is removed from the set of attributes.

Variable name: the name of a variable that contains something other than an instance of the class under test. For, example in `testConcurrentGetMachines` from `Sentinel`, `success` is the name of a variable that contains a instance of another classes (`appInfo`) that is removed from the set of attributes.

Argument: an argument passed to a non-focal method. For example, in `test_disableCookieManagement` from `Webmagic`, `cookie` is a method argument from a non-focal method that is removed from the set of attributes.

Method call: a call to a method that is declared by a class other than the class under test. For example, in `testRemove` from `Webmagic`, `isFalse` is a call to a method declared by another class that is removed from the set of attributes.

Figure 4.17 shows the number of times each type of removal occurred as a stacked barchart in the same format as Fig. 4.16. Like for addition, removal also appears to be a personal choice. For example, while Participant A often preferred to remove Parameters, Participant B often preferred to remove Method calls. Again, this suggests that a fully automated name generation approach is difficult and a cooperative approach that works with developers may be preferred.

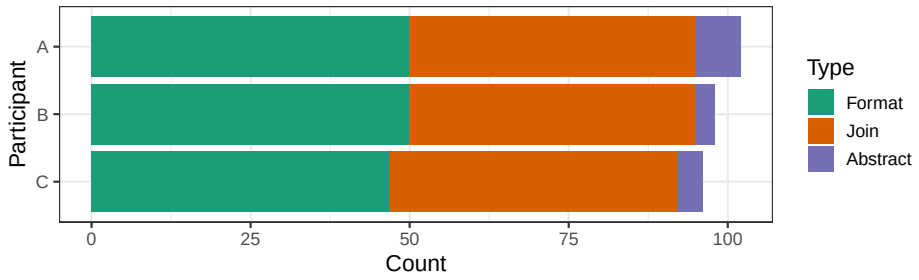


Figure 4.18: Proportion of type of attribute modifications.

4.4.3.3 Modification

Modification occurs when a participant modifies existing words or phrases in the set of attributes. As a result of the coding process, we identified the following types of modification:

Format: modify a necessary, but verbose attribute to add or remove pieces from it. For example, a leading or trailing "test" is often added to an attribute and syntactic elements such as keywords (e.g., `new`) or parenthesis are often removed. More specifically, we found an attribute —`new StandardCategoryURLGenerator()` generated from `JFreeChart`, a leading "test" is appended, and the `new` keyword and parentheses at the end of the constructor are removed. So the resulting attribute is: `testStandardCategoryURLGenerator`.

Abstract: merge two or more attributes to remove duplication. For example, `getWidth` and `getHeight` would be abstracted to `WidthHeight` and `getWidth` and `setHeightWidth` would be abstracted to `Width`. More specifically, we found two attributes—`putAsync` and `getAsync` generated from `Redisson` are abstracted to `Async`; another two attributes—`providerInvoker` and `consumerInvoker` generated from `Sentinel` are abstracted to `ProviderConsumerInvoker`.

Join: the final set of attributes are joined into one as a name for the test. For example in Fig. 4.14, the final set of attributes—`Cache` and `Async` with a prefix `test` are joined together as the final name—`TestCacheAsync`.

Figure 4.18 shows the number of times each type of modification occurred as a stacked barchart in the same format as Figs. 4.16 and 4.17. For example, Participant A made 50 Format modifications, 45 Join modifications, and 7 Abstract modifications.

Unlike for additions and removal, this data shows that modifications are much more consistent across participants. This suggests that, unlike for the other types of transformation, it is feasible to fully automate this part of the process of transforming unique attributes into unit test names.

4.5 Approach: Generation of Uniqueness-based Test Names

Based on what we learned from analyzing the transformations made by the participants, we designed the uniqueness-based name generation approach as shown in Fig. 4.19. As the figure shows, the input to the approach is a $\langle Test,\ Attributes \rangle$ pair where Test is the target test and Attributes are the unique attributes of the target test generated by the uniqueness-identification approach. The output of the approach, a name for the input test, is generated in two main steps.

4.5.1 Step 1: Additions and Removals

The purpose of the first step is to allow users of the approach to modify the set of attributes. Because we learned that additions and removals are (1) less frequent than modifications, and (2) a personal choice we made this step both optional and interactive. A user is shown the test body and the set of attributes and asked if there is anything they would like to add or remove. If they chose to make any changes, the updated set of attributes is passed to the second step.

4.5.2 Step 2: Modifications

The purpose of the second step is to modify the possibly updated attributes in order to produce a name. Unlike the first step, this step is fully automated. First, each attribute is stripped of syntactic elements like `new` or parentheses are removed. Second, the formatted attributes are abstracted to merge similar attributes in two ways. The first way to abstract the attributes is to check whether there are common

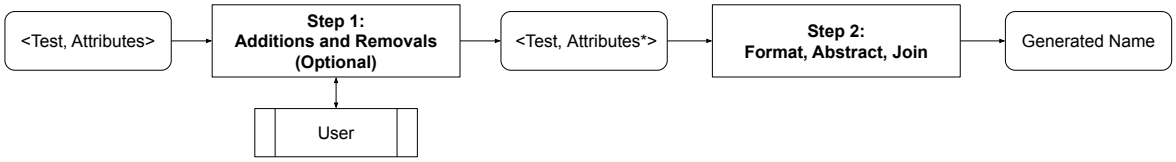


Figure 4.19: Overview of approach.

prefixes or suffixes among subsets of the attributes. To handle attributes that have a common prefix or suffix, the approach scans the attributes and grouping attributes with common prefixes and suffixes together. Then within each group, the shared text is merged into one word or phrase and the remaining text is combined without any connectors. For example, `providerInvoker` and `consumerInvoker` are abstracted to `ProviderConsumerInvoker`. The second way to abstract the attributes is to check whether there are common method pairs (e.g., get/set, put/get, etc.) among subsets of the attributes. To handle the method pairs, the approach scans the attributes and grouping attributes that contain the elements of the method pairs. The complete list of method pairs is available online [66]. Then within each group the shared method pair is deleted and the remaining text is combined without any connectors. For example, `put-Async` and `getAsync` are abstracted to `Async`. Finally, the approach joins the modified attributes by their execution order in the test body as a descriptive name.

4.6 Evaluation: Generation of Uniqueness-based Test Names

To evaluate our approach, we conducted an empirical evaluation of the names it generates. Because the perceived quality of a name is subjective, it is common to use qualitative methods, such as surveys, in this type of work [114, 79, 119, 28, 103]. In this evaluation, we considered the names generated by our approach and several alternative approaches, including the original, developer-written test names. Each name was given to a participant who was asked to assess the name with respect to several properties such as completeness and conciseness and to give an overall judgement of its quality.

Based on the participants' responses, we answered the following research questions:

(1) RQ1: Does a participant's level of familiarity with a test impact their opinions about its name?

(2) RQ2: Are there differences between a participant's opinions of conciseness and completeness for the names generated by the approaches?

(3) RQ3: Can completeness and conciseness capture the acceptability of a test name? If not, do the participants consider other qualities?

(4) RQ4: How to understand completeness and conciseness in the context of unit test names?

We first chose to investigate whether a participant's level of familiarity with a test impacts their opinions about its name. This is important since it determines whether we can treat a participant's responses uniformly or need to keep them separate in the remainder of the evaluation. Ultimately, we want to generate descriptive names, but the definition of descriptive is hard to quantify and can be significantly different under various real-world scenarios. For example, from our sanity checks with the participants after previous empirical studies, they often have a different set of principles when descriptively naming JUnit tests for different types of Java projects. Moreover, we mentioned the same concept in Section 4.3 for "there is not a single 'correct' answer" and Section 4.5 for building a cooperative name generation approach. Therefore, we used standard metrics that are not only related to the descriptiveness of JUnit test names but are commonly accepted by different developers under different scenarios. Second, based on the metrics from a well-known study [103] and our work experience, we chose to look at completeness and conciseness since they are standard metrics commonly accepted by developers. Completeness is important because it measures whether necessary information is included. Documentation that is incomplete may be missing critical information, which significantly reduces its effectiveness [70, 20]. Conciseness is important because it measures whether unnecessary information is included. Documentation that contains extraneous information may waste developers' valuable time

and effort of reading through them [83, 103]. Because test names are an important form of documentation, they should be should be both complete and concise. However, completeness and conciseness do not necessarily capture all of the things that are looked for in a test name. Therefore, the purpose of RQ3 is to capture other qualities that may influence the participants' opinions about the acceptability of test names. Finally, the purpose of RQ4 is learn more about any such qualities by directly asking about the rationales the participants use when naming tests. The remainder of this section describes our participants and how we recruited them; the test names they were asked about; the alternative approaches we considered; and a discussion of the results for each research question.

4.6.1 Participants and Subjects

As mentioned previously, our primary evaluation instrument is to survey participants. This subsection describes our selection criteria and process. As participants, we want professional developers who are experienced with a project. Choosing such developers has several benefits. First, they will be unfamiliar with our previous research or the current studies in this paper. In that case, we can eliminate the possibility of favoritism towards our approach and collect unbiased responses from them. Second, prior work has shown that students are not an analogue for professional developers [19]. This matters to us for two main reasons. For one, while prior work has used students to evaluate aspects of naming tests (e.g., [119, 28, 115]), no one has yet considered experienced developers. As domain knowledge and level of familiarity with a project may influence perception of names, the validity and generalizability of the evaluation may be impacted if we were to use students.

In order to recruit participants that meet our criteria, we started with finding people rather than projects. Experience with prior evaluations has shown that "cold calling" or unsolicited requests have a very low response rate. Rather than attempt this again, we chose to leverage our professional network (i.e., previous work experience and connections at top-tier technology companies) to meet with team leads or associate

Table 4.9: Considered projects for evaluation.

Project	Version	LoC	# Tests
Guava	368c337	400,801	13,962
Guice	9b371d3	183,049	1,280
Octane-gocd-plugin	f3cc22b	6,190	26
Partner Center SDK	cb7c53f	51,490	261
Robodriver	8286b8e	5,572	82

managers. If they agreed to work with us, we asked them to nominate a senior, staff, or principle developer from a project they oversee. If their nominee agreed, they, along with their project, became a participant. As a result of this process, we recruited a group of 5 professional software developers (i.e., senior to principle level), one from each of the projects shown in Table 4.9.

4.6.2 Considered Approaches

To compare the generated names from our approach with the alternatives, we implemented a proof-of-concept prototype of our uniqueness-based, name generation approach to be used in the empirical evaluation. This prototype of our approach is also available online [63]. When using the prototype of our approach, the names that were generated for this evaluation did not have any additions or removals in their name generation process (i.e., this part was disabled). We decided to implement this for removing a source of bias and the consistency in what the participants assessed. It is considered as a worst-case scenario from the prototype's perspective since an important part of functionality is disabled. We also considered three alternative name generation approaches: the original, developer-written names, Høst's approach [54], and Code2vec [10]. We chose the original names because they were manually written by developers experienced with the project. As such, they are the closest we have to "correct" names. The other approaches were selected because they represent two types of name generation strategies. We chose Høst's approach because it is well-studied, rule-based approach. Because there is no existing implementation of Høst's

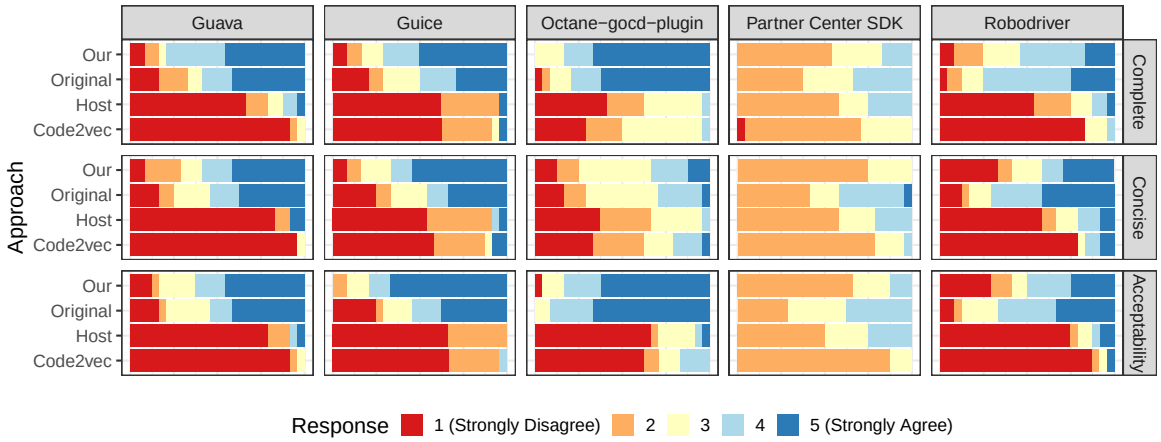


Figure 4.20: Participant agreement with completeness, conciseness, and acceptability statements.

approach, we created our own. To do this, we used the relevant naming rules from Høst's publication, which describe naming bugs and name-specific rules. We used their examples of poor names and the corresponding replacement names to verify the validity of our implementation. This implementation of Høst's approach is available online [64]. We chose Code2vec because it is a more modern, machine learning-based approach [10]. The implementation of Code2vec we used is a Commit-c98e8f7 [112].

4.6.3 Data Collection

Because evaluating a test name is relatively time consuming, it is infeasible to assess all 15,611 tests from the considered projects. Instead, we created a custom set of 24 tests for each participant. A pilot study with the authors indicated that evaluating a test would take a little over a minute, and 30 minutes is a reasonable time commitment. Each participant's test set contains 12 tests randomly chosen from their project and 3 tests randomly chosen from each other project (12 total). Including both tests from their project and from other projects allows us to assess whether their opinions differ based on their level of familiarity.

Each participant was given a survey based on their test set. For each test,

the participant is first shown the test body. Second, the participant is asked to indicate their level of agreement with the statements: "⟨*name*⟩ is complete", "⟨*name*⟩ is concise", and "⟨*name*⟩ is acceptable" where ⟨*name*⟩ is the name generated by each approach for the test body. Their responses are collected on a Likert scale from 1 to 5, where 1 indicates Strong Disagreement and 5 indicates Strong Agreement. Finally, we asked the participant to write a few sentences to further explain their responses. In particular, we asked them to detail the naming rationales behind their choices. There was no time limit for the survey, so participants were able to take as much time as they wanted to answer each question.

Figure 4.20 shows the collected responses for the completeness, conciseness, and acceptability questions in a series of 15 barcharts, one for each combination of participant/project (columns) and statement type (rows). In each barchart, the y-axis shows each considered approach, the x-axis shows the percentage of responses in each Likert category, and the color of each bar indicates the response on a diverging scale from red (Strong Disagreement) to blue (Strong Agreement). For example, the top-most, left-most barchart shows the responses from the participant who developed `Guava` when asked about the completeness of the test names. As the chart shows, the participant's response when asked about completeness was Strongly Agree ≈46 % of the time for our approach, ≈42 % of the time the original names, ≈4 % of the time for Høst's approach, and ≈0 % of the time for Code2vec. Similarly, the same participant's response when asked about completeness was Strongly Disagree ≈8 % of the time for our approach, ≈16 % of the time the original names, ≈66 % of the time for Høst's approach, and ≈91 % of the time for Code2vec.

4.6.4 RQ1: Does a Participant's Level of Familiarity with a Test Impact Their Opinions of Its Name?

The purpose of our first research question is to investigate whether a participant's level of familiarity with a test impacts their opinions about its name. The outcome of this question determines if we can treat all a participant's responses uniformly

in the rest of this evaluation; if so, we no longer need to separate this evaluation by level of familiarity. To check for a difference in the responses, we performed a series of Mann-Whitney Wilcoxon tests [32, 88]. We chose to use the Mann-Whitney Wilcoxon test because we have one nominal variable (own project or not), one measurement value (the participant's responses), and we do not know whether our data are normally distributed. We chose an alpha (α) of 0.05, and used R version 4.1.1's implementation of the test (i.e., wilcox.test). The full test results are available online [65].

In total, we ran 15 tests results: 3 types of statement (i.e., complete, concise, and acceptable) $\times$ 5 projects. The resulting p-values range from $\approx$0.11 to $\approx$0.94. For example, the p-values for the tests comparing the participant from Guava's responses is $\approx$0.49 for complete, $\approx$0.87 for the concise statements, and $\approx$0.74 for acceptable. Because all of the p-values are above 0.05, we conclude that there is not a statistically significant difference between the responses for each participant's own project and for other projects. This means that it is unlikely that a participant's level of familiarity with a test impacts their opinions of it completeness, conciseness, and acceptability. Therefore, we treat all of the responses uniformly in the rest of this evaluation rather than separate them by familiarity.

4.6.5 RQ2: Are There Differences between The Participant's Opinions of Conciseness and Completeness for The Names Generated by The Approaches?

The purpose of our second research question is to investigate whether there are differences between the participant's opinions of conciseness and completeness for the names generated by the considered approaches. To answer this research question, we use the data presented in the charts in the top two rows in Fig. 4.20. Based on a visual observation of these charts, we can make several observations:

First, it appears that the overall distribution of responses is similar for conciseness and completeness across all participants/projects. Second, the participants' opinions about the original names and the names generated by our approach are similar

and generally positive (i.e., the majority of the responses are on the Strongly Agree side of the scale) while the opinions for the names generated by Høst's approach and Code2vec are also similar but generally negative (i.e., the majority of the responses are on the Strongly Disagree side of the scale). The only outlier to these trends are the responses from the participant from `Partner Center SDK` shown in the fourth column. Overall, this participant's responses are more neutral (i.e., close to the middle of the scale) and there is not a distinct bifurcation between the responses for our approach and the original name and the responses for Høst's approach and Code2vec. We believe that these differences are due to several factors. First, `Partner Center SDK` is a Java SDK, rather than a utility library or plugin, which might necessitate a different set of test expectations. Second, this project is developed to support different programming languages and technologies (i.e., ASP.NET, REST), so the developers may bring some testing conventions used for other languages. Because all of the name generation approaches are geared toward Java JUnit tests, they might not meet all of the expectations from the other domains.

To further investigate our visual observations, we again performed a series of Wilcoxon rank sum tests. In this case though, we used pairwise tests with Bonferroni correction. Pairwise tests are necessary as the nominal value (the considered approaches) has more than two categories and Bonferroni correction adjusts for performing multiple tests. We again chose an alpha (α) of 0.05, and used R version 4.1.1's implementation of the test (i.e., pairwise.wilcox.test). The full test results are available online [65].

In total, we ran 10 tests: 2 statement types (complete and concise) $\times$ 5 projects. Each test produced 6 p-values, one for each distinct combination of considered approaches (i.e., $\binom{4}{2}$), for a total of 60 p-values). The resulting p-values range from $\approx 1.1 \times 10^{-8}$ to ≈ 1 with 28 greater than 0.05 and 32 below 0.05. These results confirm our visual observations. For all participants, the names generated by our approach are viewed as comparable (i.e., equivalent) to the original test names in terms of completeness and conciseness. Conversely, the names generated by Høst's approach and

Code2vec, while comparable to each other, are viewed as significantly less concise and complete by the participants. Overall, this a positive result for our approach. It is capable of generating test names that are as complete and concise as the original names.

4.6.6 RQ3: Can Completeness and Conciseness Capture The Acceptability of a Test Name? If not, Do the Participants Consider Other Qualities?

The purpose of RQ3 is to investigate the participants' opinions about the overall acceptability of the test names. While completeness and conciseness are important qualities, it is possible that the do not capture all of the qualities that are important when naming tests. By asking about acceptability directly, we can determine if there are important aspects of a test name that are not captured by completeness and conciseness.

To answer this research question, we use the data presented in the charts in the bottom row in Fig. 4.20. Based on a visual observation of these charts, the responses for acceptability appear to be similar to the responses for completeness and conciseness. To investigate this observation further, we used a procedure similar to the one used for RQ2: a series of pairwise Wilcoxon tests with Bonferroni correction ($\alpha = 0.05$, full test results are available online [65]). In this case, the resulting 30 p-values (1 statement types (acceptability) $\times$ 5 projects $\times$ 6 distinct approach combinations) range from $\approx 2.3 \times 10^{-8}$ to ≈ 1 with 12 p-values above the 0.05, and 18 below the 0.05. Again, results of the statistical tests confirm our visual observations. Our approach is similar to the original names; Høst's approach is similar to Code2vec; and both our approach and the original names are distinct from Høst's approach and Code2vec. This result is also positive for our approach: the names it generates are similar to the original names in terms of acceptability and are significantly better than the names generated by the alternative approaches. Moreover, the data suggests that, because the overall

distributions are similar, there are no qualities beyond completeness and conciseness that a serious concern when naming tests.

4.6.7 RQ4: How to Understand Completeness and Conciseness in The Context of Unit Test Names?

The purpose of our last research question is to better understand completeness and conciseness in the context of unit test names. While the RQ3 suggests that completeness and conciseness are the primary properties that a test name generation approach should consider, it is not clear what concrete aspects of a test map to completeness and conciseness. To investigate this research question, we used the participants' free response answers where we asked them specifically about the rationales behind their choices. We used a combination of axial and selective coding to analyze the responses along two categories: *completeness* and *conciseness*.

As a result of this process, we concluded several things about the participants' views on completeness and conciseness with respect to test names: A *complete* test name should include:

the function name that is being tested and what aspect of the function is tested. Two of the participants mentioned this rationale. Their judgement of completeness is heavily influenced by if a name contains the tested function name and what aspect of the function is tested (e.g., focal method and its testing scenario).

the class name (i.e., class under test). Two of the participants mentioned this rationale. Their judgement of completeness is heavily influenced by if a name contains the name of the class under test. They stated that having the name of the class under test can help people locate the class if there is a bug found in the process of unit testing.

the assertions in the test body. Two of the participants mentioned this rationale. Their judgement of completeness is heavily influenced by if the presented name

contains the test's assertions. They explained their reasoning behind this rationale is to use the final state and context in the assertions to provide descriptive test names.

the actual behavior of the test case. One of the participants mentioned this rationale. Their judgement of completeness is heavily influenced by if the presented name contains the actual behavior of the test case. The actual behaviors of the test could be various, such as calling a specific method to test a software feature (e.g., method calls), testing a certain parameter in a series of assertions (e.g., assertion's parameters), or initiating an object from the class under test (e.g., variables).

and a *concise* name should:

be descriptive, not generalized. Three of the participants mentioned this rationale. Their judgement of a test name being concise is heavily influenced by if the presented name is descriptive.

summarize the test and exclude irrelevant information. One participant mentioned this rationale. Their judgement of a test name being concise is heavily influenced by if the presented name can summarize the context of the test and exclude irrelevant information.

Overall, the participants' have a consistent view of what concrete aspects of a name contribute to it being complete and concise. These views also help explain why the names generated by our approach score highly while the names generated by Code2vec and Høst's approach perform poorly. First, the names generated by our approach incorporate with the views of completeness. For *the function name that is being tested and what aspect of the function is tested*, our name generation approach works well with respect to this view as it is designed to extract the focal method (function name that is being tested) and its parameters (what aspect of the function) from the test body. For *the class under test name*, our name generation approach works

well with respect to this view as it is designed to extract the name of the object under test (class name from the class under test) from the test body. For *the assertions in test body*, our name generation approach works well with respect to this view as it is designed to extract both the assertion calls and their parameters from the test body. For *the actual behavior of test*, our name generation approach works well with respect to this view as it iterates through all unique attributes (method calls, assertions, and variables) from the test body to capture the behavior of the test. Second, the names generated by our approach also incorporate with the views of conciseness. For *be descriptive*, the goal of our name generation approach directly incorporates with this high-level view, which is to provide descriptive test names. For *summarize the test and exclude irrelevant information*, the extracted attributes of a test sometimes do contain redundant information, so this view is not always met. However, it is not an issue for our name generation approach. Our approach have a Removal step to exclude the unnecessary attributes when generating names, and its prototype implementation allows users to manually inspect the existing attributes and remove any of them if deemed necessary [63]. Additionally, our approach summarize every attribute of the test when generating descriptive name as part of the Abstract step. Overall, the generated names from our approach can support this view and produce concise test names that meet developers' needs.

Existing machine learning and model-based approaches like Code2vec can sometimes capture the uniqueness of test, but are not focused on that. Therefore, their approaches are less successful in generating test names that can meet developers' needs (i.e., a name should be descriptive - complete and concise). Existing rule-based approaches like Høst's approach can occasionally produce descriptive names, but are highly relied on the robustness of their naming rules. If their naming rules fail, there could be important information missing in their names or irrelevant information added to their names. Moreover, because their naming rules are often strict and programmatic, there is no obvious way to exclude irrelevant information without violating one or more rules. Therefore, the generated names from their approaches are less likely to

be complete and concise at the same time.

Chapter 5

SUMMARY OF CONTRIBUTIONS AND FUTURE WORK

5.1 Summary of Contributions

In this dissertation, we focus on providing descriptive names that meet developers' approval for JUnit tests. Specifically, we present our dissertation work can help developers reduce the burden of finding non-descriptive names in existing JUnit test suites and semi-automatically generate descriptive names that they can use to replace them. In this chapter, we discuss the contributions of this dissertation in summary.

Chapter 1 first introduces the prevailing naming problem in unit tests and our motivation of this dissertation work. Second, it discusses several types of well-known and studied existing techniques to solve the naming problem and describes why those techniques would fail when providing descriptive names with developer's approval. Finally, it introduces our research goal and illustrates the overview about each component of this dissertation work.

Chapter 2 discussed the existing works related to this dissertation. First, this section presents several well-known existing work about detecting mismatches between test/method names and their bodies, or provide suggestions to improve their names. Second, it discusses the current research of automated generation of unit test names in several directions. One direction is to generate better test name by using predefined rules, neural probabilistic language model, or natural-language program analysis. The other direction is automatically providing renaming with better naming by lookup functions or machine learning. Third, it describes several types of research such as general program analysis, automated generation of complete test suite and debugging. For general program analysis, it discusses how existing techniques use program analysis to

analyze code or test code. For general program analysis, it discusses how existing techniques use program analysis to analyze code or test code. Last, this section describes several types of related research: natural language-based program analysis, usage of formal concept analysis in software engineering, investigation of developers' focus in software development, and several recent studies about naming.

Chapter 3 describes a new, pattern-based approach that can help developers improve the quality of test names of JUnit tests by making them more descriptive. It does this by detecting non-descriptive test names and in some cases, providing additional information about how the name can be improved. Our approach is assessed using an empirical evaluation on 34,352 JUnit tests. The results of the evaluation show that the approach is feasible, accurate, and useful at discriminating descriptive and non-descriptive names with a ≈95 % true-positive rate.

Chapter 4 describes two primary components: identification of unique attributes and generation of uniqueness-based test names. The first component is a novel approach to extract the attributes of a given test that make it unique among its siblings. The second component is a uniqueness-based JUnit test name generation approach based on the unique attributes identified by the first component, and it can generate descriptive names that meets developers' needs. Comparing with 3 alternative approaches, the generated names from our approach are similar to the original names and outperform other alternative approaches in terms of completeness, conciseness, and acceptability judged by 5 professional developers.

Finally, we anticipate that our research can help developers to locate non-descriptive names in existing test suites and semi-automatically replace them with more descriptive names that fit their needs.

5.2 Future Work

First, one possible future work to handle any threat to validity is an adaptive approach with code profiling. For the definitions of adaptive and code profiling, this

name generation approach should have the ability to achieve self-learning (i.e., adaptive) from a developer's past unit tests, coding style, and personal preferences about naming. Based on the learnt knowledge, it then creates a code profiling for this particular developer and generate descriptive names that perfectly fits their needs. The vast variety of unit tests, coding styles, and personal preferences of developers make it look like a "Mission Impossible". Because of such complexity, a possible future work is to build a reinforcement learning-based approach similar to Google's AlphaCode [75] to solve all kinds of naming requirements.

Second, another possible future work is to propose a large-scale survey towards a large group of professional developers. We have an empirical evaluation with 5 professional developers for our uniqueness-based name generation approach. However, because of the limitation of time and resources, our empirical evaluation only include 5 projects/professional developers. A more comprehensive study should invite at least 25 projects/professional developers (i.e., at least 5 times more than ours). The result of such a study might be able to find more interesting practices that developers used to name a test, and it can also help us to improve our current approach to a great extent.

BIBLIOGRAPHY

[1] S. L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. In *Proceedings of the International Conference on Program Comprehension*, pages 156–159. IEEE, 2010.

[2] N. J. Abid, J. I. Maletic, and B. Sharif. Using developer eye movements to externalize the mental model used in code summarization tasks. In *Proceedings of the ACM Symposium on Eye Tracking Research & Applications*, pages 1–9, 2019.

[3] N. J. Abid, B. Sharif, N. Dragan, H. Alrasheed, and J. I. Maletic. Developer reading behavior while summarizing java methods: Size and context matters. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 384–395, 2019.

[4] N. Alhindawi, N. Dragan, M. L. Collard, and J. I. Maletic. Improving feature location by enhancing source code with stereotypes. In *2013 IEEE International Conference on Software Maintenance*, pages 300–309. Ieee, 2013.

[5] Alibaba. Sentinel: Throttling framework for distributed systems. https://developer.alibabacloud.com/opensource/project/sentinel, 2021. Accessed: 2021-10-07.

[6] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293. ACM, 2014.

[7] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 38–49. ACM, 2015.

[8] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. PMLR, 2016.

[9] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[10] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[11] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic. An approach to automatically assess method names. *arXiv preprint*, 2022.

[12] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[13] Andrew Trenk. Testing on the Toilet: Writing Descriptive Test Names. http://googletesting.blogspot.com/2014/10/testing-on-toilet-writing-descriptive.html, 2015. Accessed: 2019-08-01.

[14] A. Arcuri and G. Fraser. Java enterprise edition support in search-based junit test generation. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 3–17, Raleigh, North Carolina, USA, 2016. Springer.

[15] A. Arcuri, G. Fraser, and J. P. Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*, pages 79–90. ACM, 2014.

[16] V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, 2014.

[17] J. Atif, I. Bloch, F. Distel, and C. Hudelot. Mathematical morphology operators over concept lattices. In *International Conference on Formal Concept Analysis*, pages 28–43, Dresden, Germany, 2013. Springer.

[18] A. Begel and H. Vrzakova. Eye movements in code review. In *Proceedings of the Workshop on Eye Movements in Programming*, pages 1–5, 2018.

[19] M. Beller, G. Gousios, and A. Zaidman. How (much) do developers test? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 559–562. IEEE, 2015.

[20] L. C. Briand. Software documentation: how much is enough? In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 13–15. IEEE, 2003.

[21] Brought to you by: Maverix, Sullis. Barbecue: Java barcode generator. https://sourceforge.net/projects/barbecue, 2018. Accessed: 2018-11-20.

[22] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of the Working Conference on Reverse Engineering*, pages 31–35, Lille, France, 2009. IEEE.

[23] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 156–165, Madrid, Spain, 2010. IEEE.

[24] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the tokenisation of identifier names. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 130–154. Springer, 2011.

[25] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux. Formal concept analysis enhances fault localization in software. In *International Conference on Formal Concept Analysis*, pages 273–288. Springer, 2008.

[26] A. Corazza, S. Di Martino, and V. Maggio. LINSEN: An efficient approach to split identifiers and expand abbreviations. In *Proceedings of the International Conference on Software Maintenance*, pages 233–242. IEEE, 2012.

[27] E. Daka and G. Fraser. A survey on unit testing practices and problems. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 201–211. IEEE, 2014.

[28] E. Daka, J. M. Rojas, and G. Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67. ACM, 2017.

[29] N. Dragan. Emergent laws of method and class stereotypes in object oriented software. In *Proceedings of the International Conference on Software Maintenance*, pages 550–555. IEEE, 2011.

[30] N. Dragan, M. L. Collard, and J. I. Maletic. Reverse engineering method stereotypes. In *Proceedings of the International Conference on Software Maintenance*, pages 24–34. IEEE, 2006.

[31] N. Dragan, M. L. Collard, and J. I. Maletic. Automatic identification of class stereotypes. In *Proceedings of the International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[32] O. J. Dunn. Multiple comparisons using rank sums. *Technometrics*, 6(3):241–252, 1964.

[33] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 71–80. IEEE, 2009.

[34] H. Eyal-Salman, A.-D. Seriai, and C. Dony. Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, pages 209–216. IEEE, 2013.

[35] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.

[36] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the SIGSOFT symposium and the European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[37] E. Gamma and K. Beck. JUnit, 2006.

[38] B. Ganter and R. Wille. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

[39] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S.-W. Lin. A neural model for method name generation from functional description. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 414–421. IEEE, 2019.

[40] M. Ghafari, C. Ghezzi, and K. Rubinov. Automatically identifying focal methods under test in unit test cases. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70. IEEE, 2015.

[41] Github. Github. https://github.com/, 2018. Accessed: 2018-12-20.

[42] B. Glaser and A. Strauss. The discovery of grounded theory. 1967. *Weidenfield & Nicolson, London*, pages 1–19, 1967.

[43] R. Godin and P. Valtchev. Formal concept analysis-based class hierarchy design in object-oriented software development. In *Formal Concept Analysis*, pages 304–323. Springer, 2005.

[44] A. Gomariz, M. Campos, R. Marin, and B. Goethals. Clasp: An efficient algorithm for mining frequent closed sequences. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 50–61. Springer, 2013.

[45] Google. Google Guava. https://github.com/google/guava, 2018. Accessed: 2018-12-20.

[46] D. Guarnera, M. L. Collard, N. Dragan, J. I. Maletic, C. Newman, and M. Decker. Automatically redocumenting source code with method and class stereotypes. In *2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, pages 3–4. IEEE, 2018.

[47] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc. Tidier: An identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process*, 25(6):575–599, 2013.

[48] N. He, P. Rümmer, and D. Kroening. Test-case generation for embedded simulink via formal concept analysis. In *Proceedings of the Design Automation Conference*, pages 224–229. IEEE, 2011.

[49] W. B. Herculano, E. L. Alves, and M. Mongiovi. Generated tests in the context of maintenance tasks: A series of empirical studies. *IEEE Access*, 10:121418–121443, 2022.

[50] W. Hesse and T. Tilley. Formal concept analysis used for software analysis and modelling. In *Formal Concept Analysis*, pages 288–303. Springer, 2005.

[51] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780, 2014.

[52] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the International Working Conference on Mining Software Repositories*, pages 79–88. ACM, 2008.

[53] E. W. Høst and B. M. Østvold. The Java programmer's phrase book. In *Proceedings of the International Conference on Software Language Engineering*, pages 322–341. Springer, 2008.

[54] E. W. Høst and B. M. Østvold. Debugging method names. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 294–317. Springer, 2009.

[55] C. Ioannou, I. Nurdiani, A. Burattin, and B. Weber. Mining reading patterns from eye-tracking data: method and demonstration. *Software and Systems Modeling*, 19(2):345–369, 2020.

[56] IssueHunt. 50 Top Java Projects on GitHub. https://medium.com/issuehunt/50-top-java-projects-on-github-adbfe9f67dbc, 2019. Accessed: 2019-06-24.

[57] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, 2016. ACL.

[58] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2013.

[59] JetBrain. IntelliJ IDEA Plugin. https://www.jetbrains.com/help/idea /plugin-development-guidelines.html, 2018. Accessed: 2018-04-04.

[60] Jianwei Wu, James Clause. Code2Seq and Code2Vec Results. https://docs.google.com/spreadsheets/d/1bnXMvaZ8JJm3xDzSq7YKMdVsKzP_icAalC6VheU7cGg/edit?usp=sharing, 2020. Accessed: 2020-08-01.

[61] Jianwei Wu, James Clause. Prototype Implementation. https://bitbucket.org/udse/findnamelite/src/master, 2020. Accessed: 2020-03-27.

[62] Jianwei Wu, James Clause. Implementation of Automated Identification of Uniqueness. https://bitbucket.org/udse/concept-analysis-plugin/src/master/, 2021. Accessed: 2021-11-06.

[63] Jianwei Wu, James Clause. Implementation of Uniqueness-based Name Generation Approach. https://github.com/Jianwei-Wu-1/NameGenerationPlugin, 2022. Accessed: 2022-09-12.

[64] Jianwei Wu, James Clause. Lite implementation of Høst's approach. https://github.com/Jianwei-Wu-1/HostsApproachLite, 2022. Accessed: 2022-09-12.

[65] Jianwei Wu, James Clause. Mann-Whitney Wilcoxon test Data. https://docs.google.com/spreadsheets/d/1DE7TMNG5fX0p_-Oh33yS0rzbnt6UW8aD_SEd5bj1jPo/edit?usp=sharing, 2022. Accessed: 2022-10-10.

[66] Jianwei Wu, James Clause. Supplemental documents. https://zenodo.org/record/6416043#.Ykysp27MJK0, 2022. Accessed: 2022-04-05.

[67] JUnit. JUnit Cookbook. https://junit.org/junit4/cookbook.html, 2018. Accessed: 2018-12-20.

[68] M. Kamimura and G. C. Murphy. Towards generating human-oriented summaries of unit test cases. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 215–218. IEEE, 2013.

[69] S. A. Kasegn and S. L. Abebe. Spatial locality based identifier name recommendation. In *2021 International Conference on Information and Communication Technology for Development for Africa (ICT4DA)*, pages 65–70. IEEE, 2021.

[70] N. J. Kipyegen and W. P. Korir. Importance of software documentation. *International Journal of Computer Science Issues (IJCSI)*, 10(5):223, 2013.

[71] A. Lakhotia. Understanding someone else's code: Analysis of experiences. *Journal of Systems and Software*, 23(3):269–275, 1993.

[72] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension*, pages 3–12, Athens, 2006. IEEE.

[73] B. Li, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Aiding comprehension of unit test cases and test suites with stereotype-based tagging. In *Proceedings of the International Conference on Program Comprehension*, pages 52–63. ACM, 2018.

[74] X. Li, W. Li, Y. Zhang, and L. Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180, 2019.

[75] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[76] P. Lison. An introduction to machine learning. *Language Technology Group (LTG)*, 1(35):291, 2015.

[77] N. Madani, L. Guerrouj, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 68–77. IEEE, 2010.

[78] K. Mens and T. Tourwé. Delving source code with formal concept analysis. *Computer Languages, Systems & Structures*, 31(3-4):183–197, 2005.

[79] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32. IEEE, 2013.

[80] L. Moreno and A. Marcus. Jstereocode: Automatically identifying method and class stereotypes in java code. In *Proceedings of the International Conference on Automated Software Engineering*, pages 358–361. IEEE/ACM, 2012.

[81] Mountainminds GmbH & Co. KG and Contributors. JaCoCo Java Code Coverage Library. https://github.com/jacoco/jacoco, 2018. Accessed: 2018-10-20.

[82] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm. A sloc counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.

[83] D. G. Novick and K. Ward. What users say they want in documentation. In *Proceedings of the 24th annual ACM international conference on Design of communication*, pages 84–91, 2006.

[84] U. Obaidellah, M. Al Haek, and P. C.-H. Cheng. A survey on the usage of eye-tracking in computer programming. *ACM Computing Surveys (CSUR)*, 51(1):1–58, 2018.

[85] Object Refinery Limited. JFreeChart. http://www.jfree.org/jfreechart, 2018. Accessed: 2018-12-20.

[86] W. Olney, E. Hill, C. Thurber, and B. Lemma. Part of speech tagging java method names. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 483–487. IEEE, 2016.

[87] Oracle. Chapter 8. Classes. https://docs.oracle.com/javase/specs/jls/se14/html/jls-8.html#jls-8.4.7, 2020. Accessed: 2020-09-10.

[88] P. A. Pappas and V. DePuy. An overview of non-parametric tests in sas: when, why, and how. *Paper TU04. Duke Clinical Research Institute, Durham*, pages 1–5, 2004.

[89] Y. Park. Software retrieval by samples using concept analysis. *Journal of Systems and Software*, 54(3):179–183, 2000.

[90] A. Peruma, E. Hu, J. Chen, E. A. AlOmar, M. W. Mkaouer, and C. D. Newman. Using grammar patterns to interpret test method name evolution. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 335–346. IEEE, 2021.

[91] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33, 2018.

[92] L. Pollock, K. Vijay-Shanker, E. Hill, G. Sridhara, and D. Shepherd. Natural language-based software analyses and tools for software maintenance. In *Software Engineering*, pages 94–125. Springer, 2009.

[93] L. Pollock, K. Vijay-Shanker, D. Shepherd, E. Hill, Z. P. Fry, and K. Maloor. Introducing natural language program analysis. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 15–16. ACM, 2007.

[94] M. Pradel and K. Sen. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):147, 2018.

[95] P. Rodeghero, C. Liu, P. W. McBurney, and C. McMillan. An eye-tracking study of java programmers and application to source code summarization. *IEEE Transactions on Software Engineering*, 41(11):1038–1054, 2015.

[96] P. Rodeghero and C. McMillan. An empirical study on the patterns of eye movement during summarization tasks. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2015.

[97] M. Schäfer, T. Ekman, and O. De Moor. Sound and extensible renaming for java. In *Proceedings of the Sigplan Notices*, volume 43, pages 277–294, Nashville, TN, USA, 2008. ACM.

[98] B. Sharif, N. Dragan, A. Sutton, M. L. Collard, and J. I. Maletic. Identifying and analyzing software design activities. In *Software Designers in Action: A Human-Centric Look at Design Work*, pages 153–174. Chapman and Hall/CRC, 2013.

[99] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 212–224. ACM, 2007.

[100] J. Singer and C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 67–76. IEEE, 2008.

[101] P. K. Singh and A. K. Ch. A note on constructing fuzzy homomorphism map for a given fuzzy formal context. In *Proceedings of the Third International Conference on Soft Computing for Problem Solving*, pages 845–855, IIT Roorkee, India, 2014. Springer.

[102] Slashdot Media. SourceForge. https://sourceforge.net, 2018. Accessed: 2018-12-22.

[103] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[104] Stanford Natural Language Processing Group. Stanford Log-linear Part-Of-Speech Tagger. https://nlp.stanford.edu/software/tagger.shtml, 2018. Accessed: 2018-04-04.

[105] A. Strauss and J. Corbin. *Basics of qualitative research techniques*. Sage publications Thousand Oaks, CA, 1998.

[106] A. A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[107] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410, L'Aquila, Italy, 2008. IEEE.

[108] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proceedings of the joint meeting of the European Software Engineering Conference and the SIGSOFT Symposium on the Foundations of Software Engineering*, pages 193–202. ACM, 2009.

[109] T. Tilley, R. Cole, P. Becker, and P. Eklund. A survey of formal concept analysis support for software engineering activities. In *Formal concept analysis*, pages 250–271. Springer, 2005.

[110] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE transactions on software engineering*, 29(6):495–509, 2003.

[111] P. Tonella. Formal concept analysis in software engineering. In *Proceedings of the International Conference on Software Engineering*, pages 743–744. IEEE, 2004.

[112] Uri Alon, Meital Zilberstein, Omer Levy and Eran Yahav. Code2vec. https://github.com/tech-srl/code2vec, 2022. Accessed: 2022-09-12.

[113] Waikato University. Weka 3: Machine Learning Software in Java. https://www.cs.waikato.ac.nz/ml/weka/index.html, 2018. Accessed: 2018-12-20.

[114] J. Wu and J. Clause. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software*, page 110639, 2020.

[115] J. Wu and J. Clause. Automated identification of uniqueness in junit tests. *ACM Transactions on Software Engineering and Methodology*, 2022.

[116] J. Yamanaka, Y. Hayase, and T. Amagasa. Recommending extract method refactoring based on confidence of predicted method name. *arXiv preprint arXiv:2108.11011*, 2021.

[117] Y. Yao. A comparative study of formal concept analysis and rough set theory in data analysis. In *International Conference on Rough Sets and Current Trends in Computing*, pages 59–68. Springer, 2004.

[118] B. Zhang, E. Hill, and J. Clause. Automatically generating test templates from test names. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 506–511. IEEE, 2015.

[119] B. Zhang, E. Hill, and J. Clause. Towards automatically generating descriptive names for unit tests. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 625–636. ACM, 2016.

[120] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 385–396, San Jose, CA, USA, 2014. ACM.

[121] H. Zhong and Z. Su. Detecting API documentation errors. In *Proceedings of the SIGPLAN Notices*, pages 803–816. ACM, 2013.

[122] Y. Zhou, X. Yan, W. Yang, T. Chen, and Z. Huang. Augmenting java method comments generation with context information based on neural networks. *Journal of Systems and Software*, 156:328–340, 2019.

Appendix A

FURTHER DISCUSSION OF TEST PATTERNS

This appendix discusses the cognitive complexity of test patterns, descriptive names, and non-descriptive names. Moreover, it also discusses if the pattern-based approach to detect non-descriptive test name is heuristic, descriptive, or based on both. In this appendix, we informally asked some of our participants (i.e., 2 student developers from the study of identifying uniqueness of tests, and 2 senior developers from the study of uniqueness-based name generation approach) for the cognitive complexity of test patterns, descriptive names, and non-descriptive names from Chapter 3. An individual, 15-minutes meeting was set up with each involved participant to discuss cognitive complexity of test patterns, descriptive/non-descriptive names, and if the pattern-based approach is heuristic, descriptive, or based on both (i.e., 60 minutes in total). In order to avoid any potential bias, this sanity check is done about three months after the participants completed our previous empirical surveys, and we only included participants who did not know about the pattern-based approach.

A.1 Cognitive complexity

We manually summarized their responses of cognitive complexity for each participant in the following sub-sections. Overall, similar to our intuition, the participants' responses show that our test patterns and descriptive names are often easy to understand, non-descriptive names are often hard to understand, and our pattern-based approach is mainly descriptive.

A.1.1 For test patterns

We presented a sub-list of both test name and body patterns to the selected participants, and these patterns were chosen by their higher frequency of appearances than other patterns. For test name patterns, we showed the complete structures of `Verb With Multiple Nouns Phrase`, `Is And Past Participle Phrase`, and `Noun Phrase`. For test body patterns, we showed the complete structures of `Try Catch`, `Normal (Generalized)`, and `All Assertion (Multiple)`. Overall, each participant spent about 30 seconds to read through and understand the purpose of each test name or body pattern. Most participants mentioned that they often met these patterns in their or other's unit tests, and some of them were looking for an approach to utilize these patterns for locating poor test names.

A.1.2 For descriptive names

We presented a small subset of descriptive names to the selected participants. For test name patterns, we showed the complete test name and body of `testGetGraphNode` from Chapter 3, `testSendACK`, and `testResponseWithErrorChunked` from Apache Tomcat. Most participants mentioned that they believe these test names are descriptive and can be understood within 30 seconds, and they only need another 30 seconds to understand the corresponding test bodies.

A.1.3 For non-descriptive names

We presented a small subset of non-descriptive names to the selected participants. For non-descriptive names, we showed the complete test name and body of `shouldThrowExceptionIfTokenIsAbsent` from Chapter 3, `testBug48839`, and `testHeader01` from Apache Tomcat. Most participants mentioned that they believe these test names are not descriptive and hard to understand. Each participant spent more than 1 minutes on reading each of these names without having a clear comprehension of the purpose of the corresponding tests, and they also needed to frequently refer back

to the test body (i.e., code). Some of them tried to import the entire project and go through all of the classes under test to figure out the purpose of the tests.

A.2 Heuristic or descriptive

At the end of each individual meeting with the selected participants, we asked them about if they feel the pattern-based approach is heuristic or descriptive and what it can improve on. Overall, based on the test patterns, descriptive and non-descriptive names presented in the meeting, 3 of the 4 participants believed that the pattern-based approach is consistent with their judgment of descriptiveness of a name. The other participant believed that it mainly follows how they would judge whether a name is descriptive. However, for some cases, this participant also mentioned that it is more heuristic but still acceptable. The participants mentioned that they would like to see an extension of test patterns to other languages like C++ or Python. Moreover, they mentioned that the current binary decision of the approach is a good fit for their daily continues integration (CI), but it would be better if it can work on other programming platforms like Visual Studio Code (VScode) or Android Studio.

Appendix B

PERMISSIONS

This appendix shows the proof of author and article reuse rights rights for our previous publications [114, 115]. All author and article reuse rights of our previous publications in this non-commercial dissertation are granted by Elsevier and Association for Computing Machinery (ACM).

B.1 Journal of Systems and Software

https://www.elsevier.com/about/policies/copyright

Author rights and article reuse description:

Author rights in Elsevier's proprietary journals	Published open access	Published subscription
Retain patent and trademark rights	√	√
Retain the rights to use their research data freely without any restriction	√	√
Receive proper attribution and credit for their published work	√	√
Re-use their own material in new works without permission or payment (with full acknowledgement of the original article): 1. Extend an article to book length 2. Include an article in a subsequent compilation of their own work 3. Re-use portions, excerpts, and their own figures or tables in other works.	√	√
Use and share their works for scholarly purposes (with full acknowledgement of the original article): 1. In their own classroom teaching. Electronic and physical distribution of copies is permitted 2. If an author is speaking at a conference, they can present the article and distribute copies to the attendees 3. Distribute the article, including by email, to their students and to research colleagues who they know for their personal use 4. Share and publicize the article via Share Links, which offers 50 days' free access for anyone, without signup or registration 5. Include in a thesis or dissertation (provided this is not published commercially) 6. Share copies of their article privately as part of an invitation-only work group on commercial sites with which the publisher has a hosting agreement	√	√

Figure B.1: Reuse for Journal of Systems and Software.

B.2 ACM Transactions on Software Engineering and Methodology

https://authors.acm.org/author-resources/author-rights

Author rights and article reuse description:

Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is notthe editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

Figure B.2: Reuse for ACM Transactions on Software Engineering and Methodology.