Automated Identification of Uniqueness in JUnit Tests

JIANWEI WU, University of Delaware, USA
JAMES CLAUSE, University of Delaware, USA

In the context of testing, descriptive test names are desirable because they document the purpose of tests and facilitate comprehension tasks during maintenance. Unfortunately, prior work has shown that tests often do not have descriptive names. To address this limitation, techniques have been developed to automatically generate descriptive names. However, they often generated names that are invalid or do not meet with developer approval. To help address these limitations, we present a novel approach to extract the attributes of a given test that make it unique among its siblings. Because such attributes often serve as the basis for descriptive names, identifying them is an important first step towards improving test name generation approaches. To evaluate the approach, we created a prototype implementation for JUnit tests and compared its output with human judgment. The results of the evaluation demonstrate that the attributes identified by the approach are consistent with human judgment and are likely to be useful for future name generation techniques.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Theory of computation →** *Program analysis.*

Additional Key Words and Phrases: unit testing, formal concept analysis

1 INTRODUCTION

Comprehension is a frequent task in test maintenance because understanding the purposes of tests is the first step towards modifying existing tests, adding new tests, or removing unnecessary tests. This task can be difficult because the necessary information is not always explicitly stated, rather it exists implicitly in the source code of unit tests and the classes under test. As an alternative to reading tens or hundreds of lines of code, summary information, written in natural language, is often provided in the form of a descriptive test name. If every test in a test suite has a descriptive name, a developer can locate a test and understand its purpose quickly by only reading its name. The collection of test names can also serve as detailed documentation for a test suite. Therefore, descriptive test names can significantly benefit test maintenance activities.

Unfortunately, prior work has demonstrated that tests often do not have descriptive names [56]. Poor names are often written by developers and widely exist in the test suites of real world projects. For example, developers often create test names for convenience (e.g., test1, testB, etc.) and ignore future maintenance needs [15, 53, 56]. In addition, during maintenance, developers may make existing test names erroneous when they change the source code of tests without changing corresponding names [30].

Existing approaches have tried to address the problem of poor names by automatically generating descriptive test names. Some of these approaches use predefined rules (e.g., [25, 56]), API-level coverage goals, or language models (e.g., [4, 15]). Other approaches attempt to solve the problem by automatically generating unit tests (i.e., both name and body) [8, 17, 55]. While these types of approaches can be successful at generating descriptive

Authors' addresses: Jianwei Wu, University of Delaware, 18 Amstel Ave, Newark, DE, USA, wjwcis@udel.edu; James Clause, University of Delaware, 18 Amstel Ave, Newark, DE, USA, clause@udel.edu.

1049-331X/2022/5-ART $15.00
https://doi.org/10.1145/3533313

names or test cases, such names often do not meet with developer approval (e.g., because the generated names do not follow the developer's preferred naming rationales, as discussed in Section 5). 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., [6, 7]). 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 [32]. In addition, these approaches are primarily geared towards general methods and have problems when applied to test name generation. For example, we have found that, due to the similarity of test bodies, these state-of-the-art approaches (i.e., [6, 7]) often generate duplicate names for tests in the same class (see [30]). While such names may make sense individually, 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.

Because of the above limitations of existing approaches, further work is needed in order to automatically generate test names that are both descriptive, take into account the differences between test names and general method names, and meet with developer approval. Our work in this direction is inspired by a belief, informed from our experiences with many tests from many projects, that a test's name is often based on what makes the test unique from its siblings—tests declared in the same class. To investigate this belief, we conducted a large-scale empirical study, considering tests drawn from 11 open-source projects, that asks whether tests are named after what makes them unique. The results of the study: (1) confirm our impression that the majority of tests are named after what makes them unique, and (2) identify additional aspects that influence how tests are named.

Based on the results from the study, we designed a novel, automated approach that can extract the attributes of a test that make it unique among its siblings. Because descriptive names are often based on such unique attributes, the approach is a crucial building block for future name generation techniques. At a high-level, the approach uses a combination of static program analysis to extract a variety of candidate attributes, identified as part of the study, from a test suite and formal concept analysis to identify which combination of attributes is unique to the test under consideration.

To evaluate the approach, we implemented it as a working prototype for applications written in Java using the JUnit testing framework. We chose these technologies as they are commonly used. Using this prototype, we conducted an empirical evaluation based on 920 tests taken from 17 open source projects hosted on GitHub. The results of the evaluation show that our proposed approach is consistent with human judgment. More specifically, this paper makes the following contributions:

(1) an empirical study to investigate if unit tests are named after what makes them unique
(2) an automated approach that identifies the unique attributes of unit tests
(3) a prototype implementation of the approach for JUnit tests
(4) an empirical evaluation that demonstrates the approach's output is consistent with human judgment and requires less effort to be transformed into descriptive test names

The remainder of this paper is organized as follows: Section 2 describes our empirical study of unit tests that motivates the uniqueness-based approach. Section 3 describes the details of the approach and the prototype implementation of the approach. Section 4 presents the empirical evaluation of the approach with a research question. Finally, Section 7 presents our conclusions and planned future work, and Section 5 discusses the related work.

2 EMPIRICAL STUDY: ARE TESTS NAMED FOR WHAT MAKES THEM UNIQUE?

Our belief is that developers often name unit tests 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.

Table 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

2.1 Experimental Subjects

To gather the tests we examined in our study, we started with the 11 Java projects shown in Table 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 [35]; and the final column, *# Tests*, shows the number of unit tests in the project. In total, these 11 projects contain 25,459 unit tests.

The first ten projects were randomly selected from the top 50 Java projects hosted on Github [27]. 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., [53, 55, 56]). 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 unit 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 [53, 56], 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 poor 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 (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 [30].

2.2 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.

2.2.1 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 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.

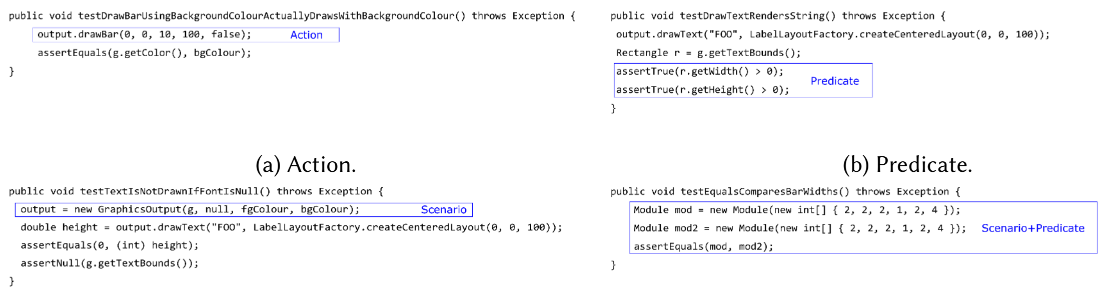
Because there is no pre-existing classification scheme for identifying what makes a test unique, we used open, axial, and selective coding to qualitatively analyze the tests [21, 45]. First, each author individually examined each of the 440 considered tests and tagged the portions of the test body that 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.

2.2.2 Selective Codes. The set of selective codes is based partly on the Java language elements [37] 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 unit tests. More specifically, we created four primary codes that correspond to the high-level structure of the test (i.e., parts of test [56]) 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 [15, 20, 53]. 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. 1 shows code examples for each of the primary codes.

2.2.3 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., [11, 39, 40]).

(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



```
public void testDrawBarUsingBackgroundColourActuallyDrawsWithBackgroundColour() throws Exception {
     output.drawBar(0, 0, 10, 100, false);
     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);
}
```

(b) Predicate.

```
public void testTextIsNotDrawnIfFontIsNull() throws Exception {
  output = new GraphicsOutput(g, null, fgColour, bgColour);
  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 });
    assertEquals(mod, mod2);
}
```

(d) Combination.

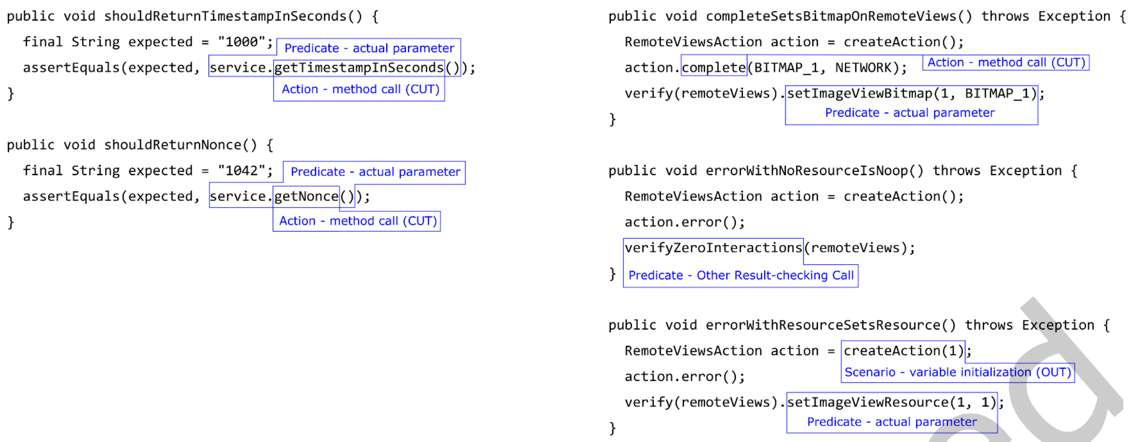
Fig. 1: Examples of primary codes.

(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. 1a 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 [56]. 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.

2.2.4 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



```
public void shouldReturnTimestampInSeconds() {
  final String expected = "1000";
  assertEquals(expected, service.getTimestampInSeconds());
}

public void shouldReturnNonce() {
  final String expected = "1042";
  assertEquals(expected, service.getNonce());
}
```

```
public void completeSetsBitmapOnRemoteViews() throws Exception {
  RemoteViewsAction action = createAction();
  action.complete(BITMAP_1, NETWORK);
  verify(remoteViews).setImageViewBitmap(1, BITMAP_1);
}

public void errorWithNoResourceIsNoop() throws Exception {
  RemoteViewsAction action = createAction();
  action.error();
  verifyZeroInteractions(remoteViews);
}

public void errorWithResourceSetsResource() throws Exception {
  RemoteViewsAction action = createAction(1);
  action.error();
  verify(remoteViews).setImageViewResource(1, 1);
}
```

Predicate - actual parameter
Action - method call (CUT)
Predicate - actual parameter
Action - method call (CUT)
Action - method call (CUT)
Predicate - actual parameter
Predicate - Other Result-checking Call
Scenario - variable initialization (OUT)
Predicate - actual parameter

(a) Test cases with tagged text from Scribejava.

(b) Test cases with tagged text from Picasso.

Fig. 2: Examples of test cases with tagged text.

(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. 1b 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.

2.2.5 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 [20]. 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 [20], and (ii) no other sibling contains the same state-changing CUT method call in its scenario, irrespective of the method arguments.

For example, Fig. 1c 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.

2.2.6 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. 1d 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.

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. 2 (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.

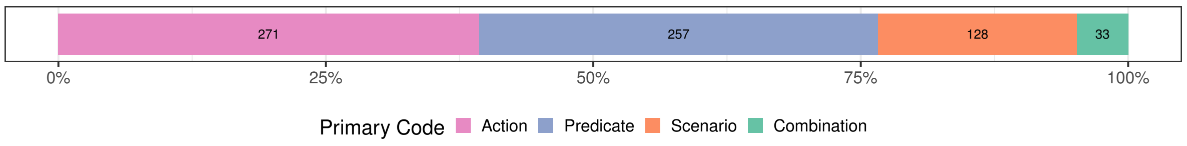


Fig. 3: Frequency of primary codes.

For example, in Fig. 2a, 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. 2b, 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: "`verify-ZeroInteractions`" because no sibling shares the test's predicate, which is to call a verification method `verify-ZeroInteractions` to check the behavior of the `remoteViews` variable. The test `errorWithResourceSets-Resource` 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 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 [30].

2.3.1 Data and Discussion. To better understand the results of the coding process we gathered some descriptive statistics. Figure 3 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 shows the frequencies of each secondary code in four horizontal 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

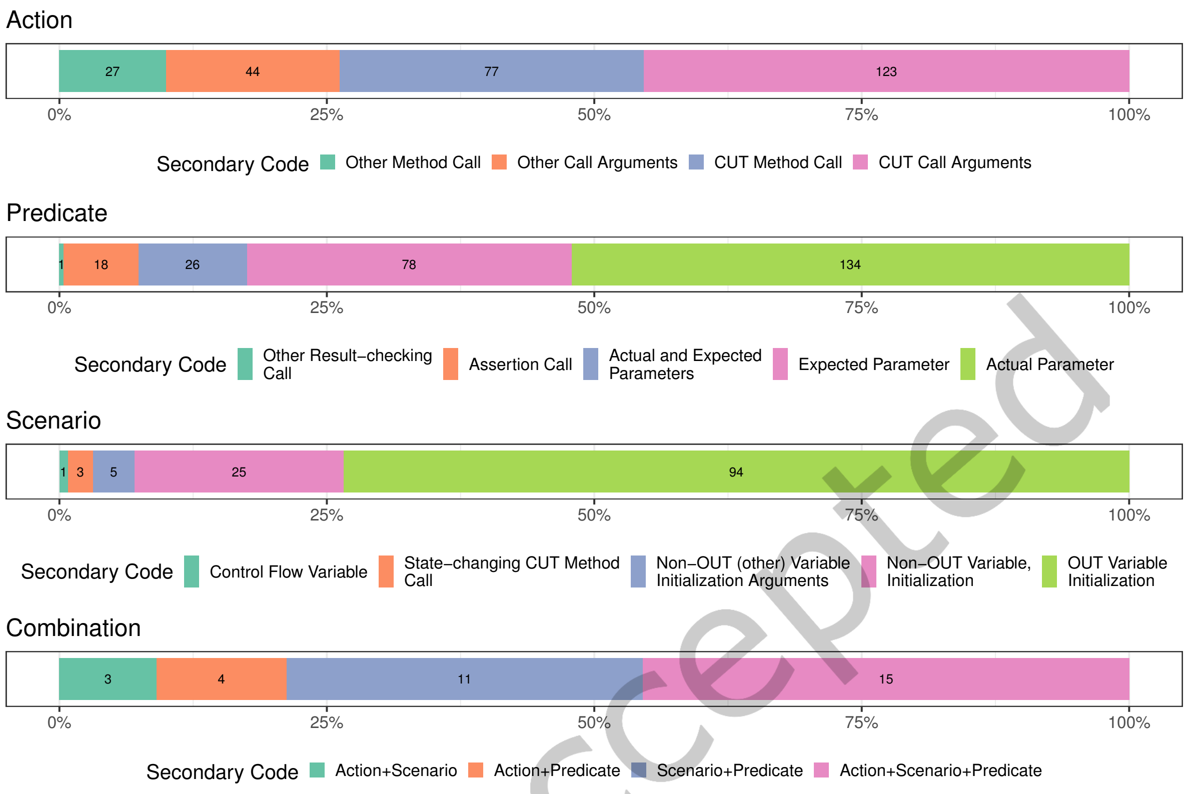


Fig. 4: Frequency of secondary codes.

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 [30].

2.4 Phase 2: Deciding Whether Unit Tests Are Named After What Makes Them Unique

The goal of the second part of our study is to investigate whether tests are named, either wholly or in part, after what makes them unique. In order to decide whether tests 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?

2.4.1 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 [16]. 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.) [30]. For example, the test name `testPostReturnsBarcodeImage` is parsed into a set of tokens: $\langle post, returns, barcode, image\rangle$. 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 $\langle put, all\rangle$ and the tokens from the tagged text are also $\langle put, all\rangle$. 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 $\langle geo\rangle$ and the tokens from the tagged text are $\langle geometry\rangle$. 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 `test-ChecksumIsNull` from Barbecue, the tokens from the name are $\langle checksum, is, null\rangle$ and the tokens from the tagged text are $\langle calculate, checksum\rangle$. 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 $\langle child, bindings, visible, parent\rangle$ and the tokens from the tagged text are $\langle get, binding\rangle$. Because the token "bindings" from the test name is the plural of the token "binding" from the tagged text, `testChildBindingsNotVisibleToParent` 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.

Table 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

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 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 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 ⟨*read*⟩, 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 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.

2.4.2 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 $\langle get, parameter, null, access, secret\rangle$. 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 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

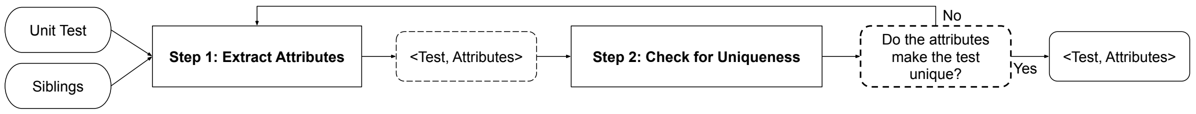

Unit Test
Siblings
Step 1: Extract Attributes
<Test, Attributes>
Step 2: Check for Uniqueness
Do the attributes make the test unique?
No
Yes
<Test, Attributes>

Fig. 5: Overview of the automated approach.

and transformed tokens are largely consistent. For example in Table 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.

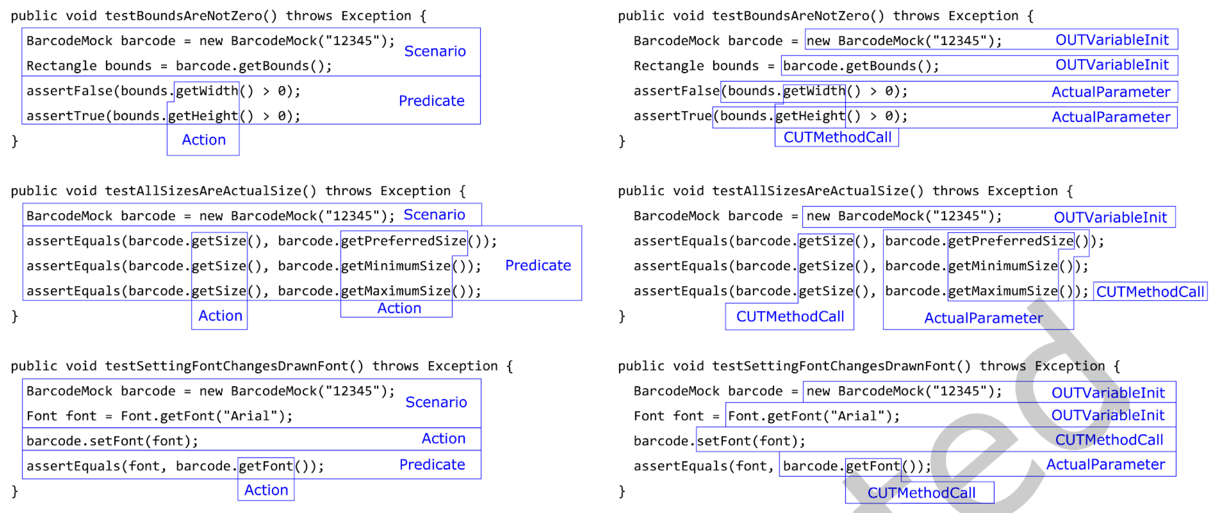
3 AN AUTOMATED APPROACH FOR IDENTIFYING THE UNIQUE ATTRIBUTES OF TESTS

Figure 5 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 2. 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.

3.1 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 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



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

public void testAllSizesAreActualSize() throws Exception {
  BarcodeMock barcode = new BarcodeMock("12345");
  assertEquals(barcode.getSize(), barcode.getPreferredSize());
  assertEquals(barcode.getSize(), barcode.getMinimumSize());
  assertEquals(barcode.getSize(), barcode.getMaximumSize());
}

public void testSettingFontChangesDrawnFont() throws Exception {
  BarcodeMock barcode = new BarcodeMock("12345");
  Font font = Font.getFont("Arial");
  barcode.setFont(font);
  assertEquals(font, barcode.getFont());
}
```

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

public void testAllSizesAreActualSize() throws Exception {
  BarcodeMock barcode = new BarcodeMock("12345");
  assertEquals(barcode.getSize(), barcode.getPreferredSize());
  assertEquals(barcode.getSize(), barcode.getMinimumSize());
  assertEquals(barcode.getSize(), barcode.getMaximumSize());
}

public void testSettingFontChangesDrawnFont() throws Exception {
  BarcodeMock barcode = new BarcodeMock("12345");
  Font font = Font.getFont("Arial");
  barcode.setFont(font);
  assertEquals(font, barcode.getFont());
}
```

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

Fig. 6: Example outputs of information matchers.

```
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);
}
```

Fig. 7: Example of state-change methods.

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. 2b, 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 [53, 56]. For example in Fig. 7, 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. 6 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. 6. 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:

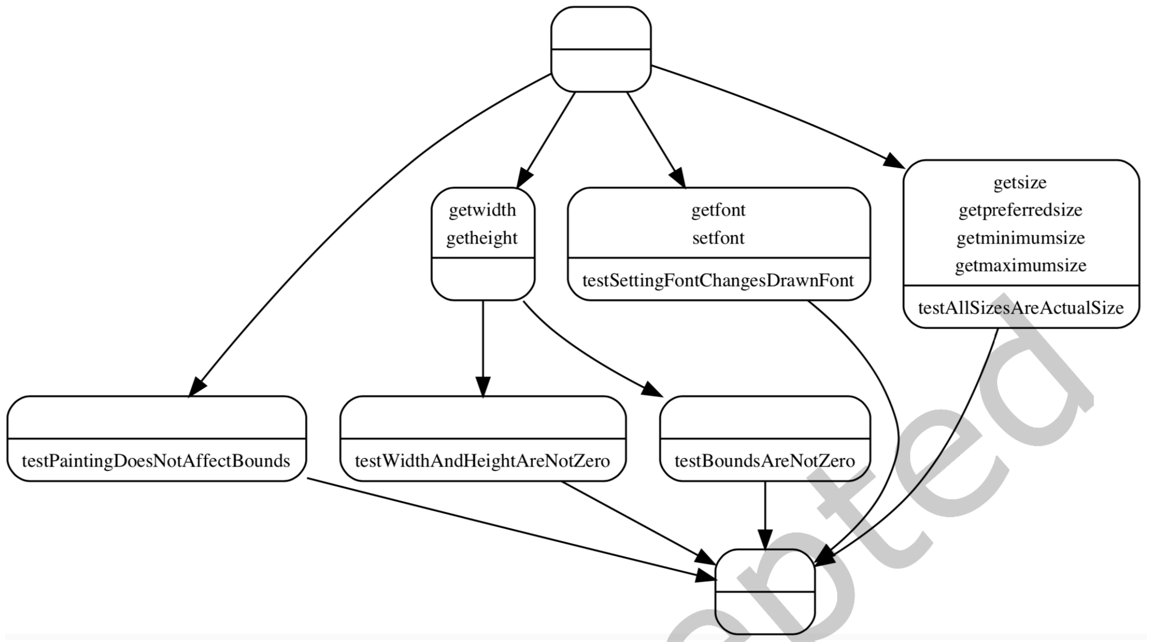
- ⟨`testBoundsAreNotZero`, ⟨`new BarcodeMock("12345")`, `barcode.getBounds()`⟩⟩
- ⟨`testAllSizesAreActualSize`, ⟨`new BarcodeMock("12345")`⟩⟩
- ⟨`testSettingFontChangesDrawnFont`, ⟨`new BarcodeMock("12345")`, `Font.getFont("Arial")`⟩⟩

3.2 Step 2: Check for Uniqueness

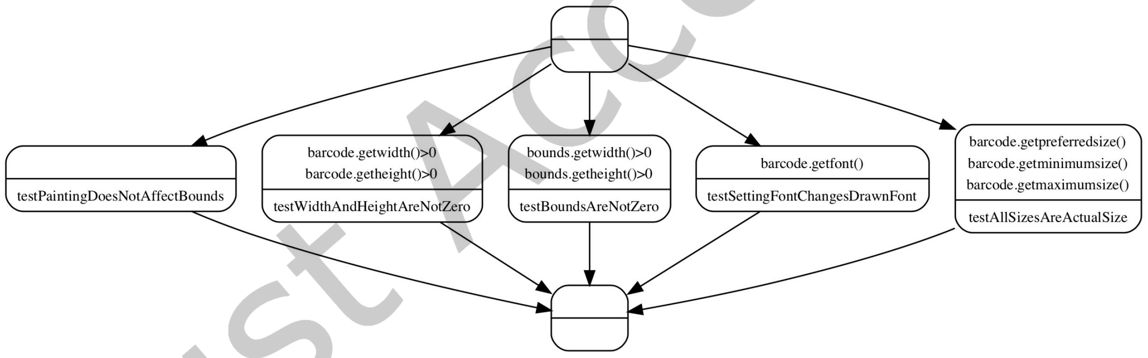
The goal of Step 2 is to identify whether the current set of attributes extracted by Step 1 make the target test unique among its siblings. This is accomplished using formal concept analysis (FCA) on the set of ⟨*test*, *attributes*⟩ pairs from Step 1 to build a concept lattice. The lattice is then analyzed 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., [49, 51]). In our approach, *objects* are the tests and *attributes* are the attributes extracted by Step 1.

As examples of a concept lattice, consider Fig. 8. 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 [50]. A formal concept is a pair consisting of a subset of objects and a subset of attributes that is closed by Galois connection [43]. 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


getwidth
getheight
getfont
setfont
testSettingFontChangesDrawnFont
getsize
getpreferredsize
getminimumsize
getmaximumsize
testAllSizesAreActualSize
testPaintingDoesNotAffectBounds
testWidthAndHeightAreNotZero
testBoundsAreNotZero

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



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

Fig. 8: Example concept lattices derived using FCA.

concept that contains all attributes, and the bottommost node is the concept that contains all objects [10]. In Fig. 8, each lattice was built for our running example from Fig. 6 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. 8a 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`.

```
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]
```

Fig. 9: Example output showing the unique attributes for two tests.

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. 8a `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. 8a 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. 8a, Fig. 8b 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. 8.

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. 9. 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`.

Table 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

4 EMPIRICAL EVALUATION

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 [29]. 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 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.

4.1 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.

4.1.1 Experimental 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 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,

Table 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

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 [30].

4.1.2 Discussion. To answer the question about whether the approach can automate our guidelines 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 2). 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 (`getFake-DefaultContext, 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."`).

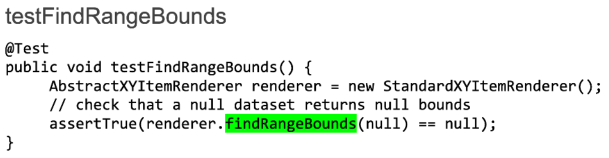
A result in the first three categories indicates cases where the approach is able to follow the guidelines that we provided for extracting unique attributes from unit tests. For tests in the Equivalent category, our approach identifies the same unique attributes as the guidelines describes, which could 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 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 `shouldHandleSchemeInsensitiveCase` from Picasso, the approach used a higher-ranked code `Action-OtherMethodCall`, but the human judgment corresponds to a lower-ranked code `Predicate-Actual-Parameter`. 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



testFindRangeBounds

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

Fig. 10: Example of a participant-labeled test.

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 `nullBitmapOptionsIfNoResizingOrPurgeable` 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-OtherMethodCallArgument` 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 a given test among its siblings. For example, for `testSlowRequest-Mode` 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, entryAndSleep-For`) 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-Other-MethodCall` 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).

4.2 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.

4.2.1 Experimental Subjects. 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. 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 [30]. 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. 10. In the figure, the part of `testFindRangeBounds` that the participant believes makes it unique, the method

Table 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

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 [30].

4.2.2 Discussion. 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 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 [3]. 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 unit 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 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 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.

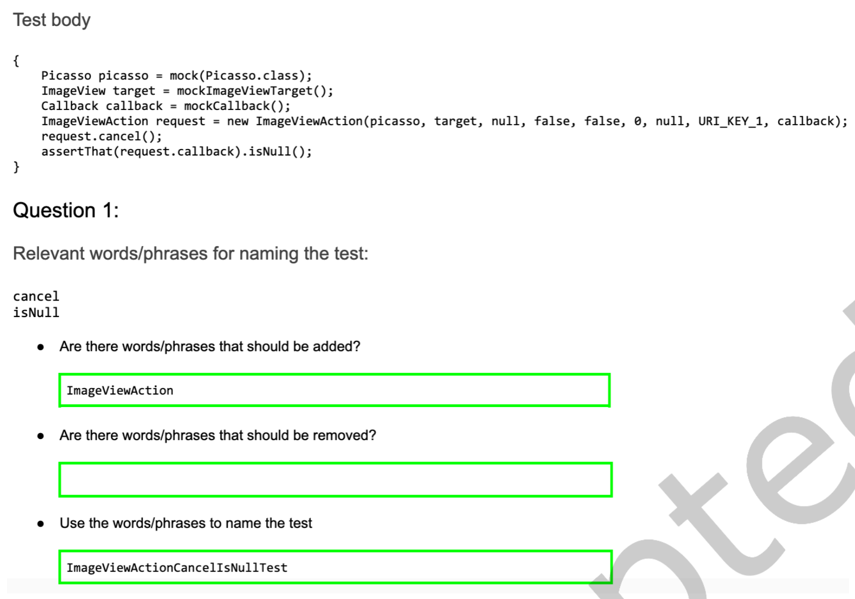
Additionally, we also looked into what kind of statements developers often agree on as being what makes the test unique. For example, some developers might prefer to select the uniqueness of test from assertions, and others might not. Overall, from our three participants, the count of assertions that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

4.3 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 [7]. 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., [5, 6, 28]). 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


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`

Fig. 11: Example of a participant response for constructing descriptive name.

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 11 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.

Each participant answered each set of questions and took approximately 8 hours in total to answer all questions. In the example shown in Fig. 11, 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.

4.3.1 Discussion. 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 12 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

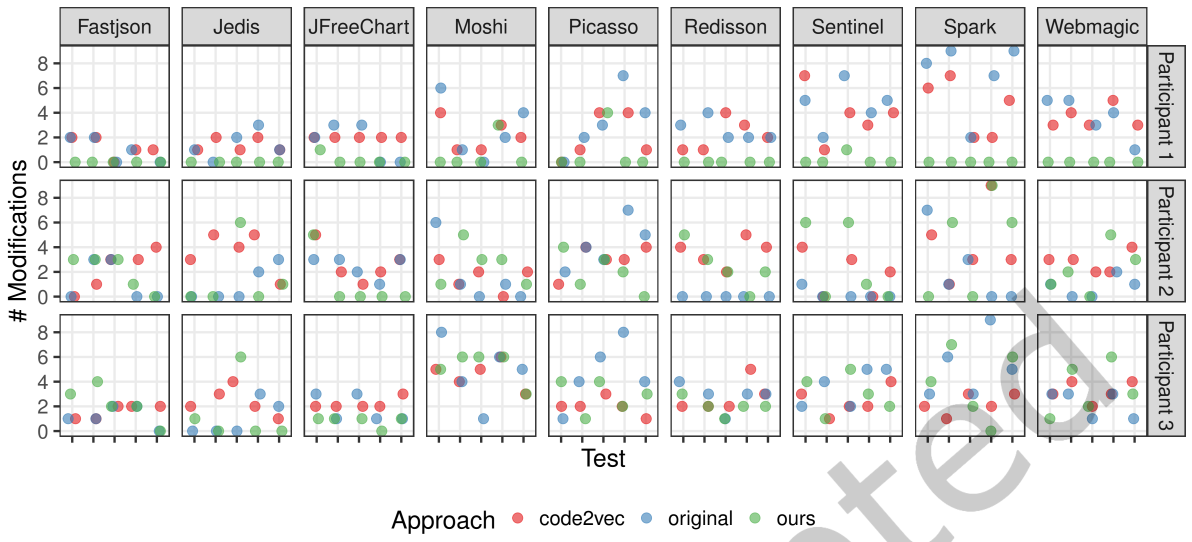


Fig. 12: Data showing the count of modifications for each approach and for each project.

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 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 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

Table 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

Table 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

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 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 7 shows the results of this calculation in the same format as Table 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.

5 RELATED WORK

In this paper, we present an automated approach to extract unique attributes from JUnit tests based on the uniqueness-based naming rationale mentioned in Section 2. This approach uses several different techniques, including static analysis, program analysis, formal concept analysis, and lattice traversal, so this section is needed to review the most closely related works that come from each field.

5.1 Generating Descriptive Names

Daka et al. introduced a test name generation technique that can summarize a series of coverage goals under the associated test suite [15]. They tried to generate descriptive names for a set of automatically generated JUnit tests [18] that are produced by an automated test generation tool named EvoSuite [17]. 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.

Schäfer et al. proposed an extensible technique that can address the renaming problem of classes, methods, and fields [41]. 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 [6, 7]. 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 [30]. 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.

Wu and Clause provided a valuable insight about how to suggest descriptive test names using information comparison, but their work is still limited to a subset of existing tests [53]. 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 [4, 25, 41, 56]. 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 [9, 12, 13, 31, 46]. 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.

5.2 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 and McMillan conducted an empirical study to discover the patterns of eye movement when developers perform summarization tasks [40]. Begel and Vrzakova 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 [11]. Abid et al. focused on building a mental cognition model to understand how developers understand codes [1]. Obaidellah et al. conducted a throughout survey on the usage of the eye-tracking technology in the research of program comprehension [36]. Abid et al. presented a eye-tracking study that shows the behavior of developers when summarizing Java methods [2]. Ioannou et al. proposed a series of reading pattern by mining the eye-tracking data for comprehending the behavior of developers [26]. 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.

5.3 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 [19, 54]. Moreover, a early survey has already been conducted to discover the possibility of solving software engineering problem by using concept analysis [49]. Many researchers focused on using FCA as an efficient method to provide solutions for various software engineering problems [14, 23, 24, 34, 38], and they used concept analysis to mock how human (i.e., developers) thinks about writing software or trace the thinking inside existing software. However, rather than applying FCA to unit tests, their techniques focused more on using FCA to understand the high-level structure of software rather than improving the naming of unit tests.

5.4 Program Analysis and Automated Test Generation

Techniques in this category are related to our work because they tried to solve some related problems (i.e., about method or test naming in general) by using program analysis. A group of researchers tried to automatically generate comments, summarization, and perform software analysis [16, 33, 44, 58]. Another group of researchers focused on analyzing an existing test suite and improving its effectiveness with different techniques [22, 42, 52, 57]. Furthermore, existing approaches also attempted to automatically generate test cases to solve the test naming problem [8, 17, 47, 48, 55]. The research goals of these techniques are essentially different from our approach, but they provide necessary references for us to build our automated, uniqueness-focused approach.

6 THREAT TO VALIDITY

While the overall results of the study and evaluation are positive, they also revealed some threats to validity that can be addressed in our planned future work in Section 7. Three threats to validity exist for the automated approach: (1) although we did our best effort to include different types of experimental subjects (i.e., unit tests), our selective codes might still not be able to capture some test cases from other Java projects, (2) the automated approach is currently not tried on other programming languages, and (3) our invited participants included students with JUnit programming experience but did not include the original developers.

For the first two threats, we plan to resolve them by adding more experimental subjects to a larger scale evaluation and try our approach on other programming languages like `CppUnit`. For the last threat, although inviting PhD students with a reasonable amount of development experience might not be sufficient for a more comprehensive study (i.e., inviting the original developers would be the most ideal method), it is a common practice in software engineering research [13, 15, 44, 55, 58]. In our planned future work, we will try to communicate with some of the original developers of the experimental subjects and invite them to a more comprehensive study and evaluation.

7 CONCLUSION AND FUTURE WORK

In this paper, we conducted a large-scale empirical evaluation, using tests extracted from 11 projects, that investigated the rationale behind naming unit tests. We found that a majority of tests are named, either wholly or in part, after what makes them unique from their siblings. This finding motivated a new, automated approach for extracting the unique attributes of JUnit tests. Such attributes can then be used as the basis for future unit test name generation techniques.

Based on the results of the empirical study, we developed the automated approach to extract the unique attributes of tests. Overall, our automated approach achieves this goal in two steps. For Step 1, the approach used the matchers for the selective codes to extract attributes of the test. For Step 2, the approach used FCA to determine if the extracted attributes can uniquely identify the test among its siblings. To evaluate our approach, we implemented it as a working prototype, and an empirical evaluation was conducted on a set of random subjects. The results of the evaluation shows that the attributes identified by the approach are consistent with human judgment and therefore are likely to useful for future name generation techniques.

For the imminent step of our planned future work, we will randomly select another set of 100 Java projects and invite a different team of human raters to perform a similar comparison between our approach and other baselines. Based on this broader range of subjects, we also plan to further investigate the difference between different types of statements from our human-labelled results, which might show us if some kinds of statements are more "stable" judged by our human raters than others. Another step is to investigate if the automated approach could be applicable to the unit tests of other programming languages like `CppUnit` or `PyUnit`.

For the rest of our planned future work, first, as we stated in Section 2, we are going to look into the less common naming rationales and try to formulate a more specific framework to cover them. Second, from the results of the empirical study, we are going to further inspect the Partial category since it takes up a majority of test names that are partially constructed after what makes the test unique. For this part, we hope to have a better understanding of the origin of other types of tokens (i.e., not the tokens that makes the test unique) in those test names, so it would be possible to upgrade our current approach to extract them as well. Last, we already gathered a sufficient amount of descriptive test names constructed by human developers as part of the empirical evaluation in Section 4. Combined with the knowledge we gained from previous steps, we plan to build a name generation tool to automatically provide descriptive test names in the same way as developers would do.

REFERENCES

[1] Nahla J Abid, Jonathan I Maletic, and Bonita Sharif. 2019. 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*. ACM, Denver, Colorado, USA, 1–9.
[2] Nahla J Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I Maletic. 2019. Developer reading behavior while summarizing java methods: Size and context matters. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. IEEE,

Montréal, Canada, 384–395.

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

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

[5] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. PMLR, New York, NY, USA, 2091–2100.

[6] Uri Alon, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. *CoRR* abs/1808.01400 (2018), 1–22. arXiv:1808.01400 http://arxiv.org/abs/1808.01400

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.

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

[9] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2014. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering* 40, 5 (2014), 502–532.

[10] Jamal Atif, Isabelle Bloch, Felix Distel, and Céline Hudelot. 2013. Mathematical morphology operators over concept lattices. In *International Conference on Formal Concept Analysis*. Springer, Dresden, Germany, 28–43.

[11] Andrew Begel and Hana Vrzakova. 2018. Eye movements in code review. In *Proceedings of the Workshop on Eye Movements in Programming*. ACM, Warsaw, Poland, 1–5.

[12] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of the Working Conference on Reverse Engineering*. IEEE, Lille, France, 31–35.

[13] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE, Madrid, Spain, 156–165.

[14] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. 2008. Formal concept analysis enhances fault localization in software. In *International Conference on Formal Concept Analysis*. Springer, Montréal, Canada, 273–288.

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

[16] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the International Working Conference on Mining Software Repositories*. IEEE, IEEE, Vancouver, Canada, 71–80.

[17] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the SIGSOFT Symposium and the European conference on Foundations of software engineering*. ACM, Szeged, Hungary, 416–419.

[18] Erich Gamma and Kent Beck. 2006. JUnit.

[19] Bernhard Ganter and Rudolf Wille. 2012. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, TU Dresden.

[20] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. 2015. Automatically identifying focal methods under test in unit test cases. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*. IEEE, IEEE, Bremen, Germany, 61–70.

[21] Barney Glaser and Anselm Strauss. 1967. The discovery of grounded theory. 1967. *Weidenfield & Nicolson, London* 17, 4 (1967), 1–19.

[22] Patrice Godefroid, Peli de Halleux, Aditya V Nori, Sriram K Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y Levin. 2008. Automating software testing using program analysis. *IEEE software* 25, 5 (2008), 30–37.

[23] Robert Godin and Petko Valtchev. 2005. Formal concept analysis-based class hierarchy design in object-oriented software development. In *Formal Concept Analysis*. Springer, Lens, France, 304–323.

[24] Wolfgang Hesse and Thomas Tilley. 2005. Formal concept analysis used for software analysis and modelling. In *Formal Concept Analysis*. Springer, Lens, France, 288–303.

[25] Einar W Høst and Bjarte M Østvold. 2009. Debugging method names. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, Genoa, Italy, 294–317.

[26] Constantina Ioannou, Indira Nurdiani, Andrea Burattin, and Barbara Weber. 2020. Mining reading patterns from eye-tracking data: method and demonstration. *Software and Systems Modeling* 19, 2 (2020), 345–369.

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

[28] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. 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)*. ACL, Berlin, Germany, 2073–2083.

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

[30] Jianwei Wu, James Clause. 2022. Supplemental documents. https://zenodo.org/record/6416043#.Ykysp27MJK0. Accessed: 2022-04-05.
[31] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th IEEE International Conference on Program Comprehension*. IEEE, Athens, 3–12.
[32] Pierre Lison. 2015. An introduction to machine learning. *Language Technology Group (LTG)* 1, 35 (2015), 291.
[33] Inderjeet Mani. 2001. *Automatic summarization*. Vol. 3. John Benjamins Publishing, Sunnyvale, CA.
[34] Kim Mens and Tom Tourwé. 2005. Delving source code with formal concept analysis. *Computer Languages, Systems & Structures* 31, 3-4 (2005), 183–197.
[35] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC counting standard. In *Cocomo ii forum*, Vol. 2007. Citeseer, Los Angeles, CA, USA, 1–16.
[36] Unaizah Obaidellah, Mohammed Al Haek, and Peter C-H Cheng. 2018. A survey on the usage of eye-tracking in computer programming. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–58.
[37] Oracle. 2020. Chapter 8. Classes. https://docs.oracle.com/javase/specs/jls/se14/html/jls-8.html#jls-8.4.7. Accessed: 2020-09-10.
[38] Young Park. 2000. Software retrieval by samples using concept analysis. *Journal of Systems and Software* 54, 3 (2000), 179–183.
[39] Paige Rodeghero, Cheng Liu, Paul W McBurney, and Collin McMillan. 2015. An eye-tracking study of java programmers and application to source code summarization. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1038–1054.
[40] Paige Rodeghero and Collin McMillan. 2015. 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)*. IEEE, Beijing, China, 1–10.
[41] Max Schäfer, Torbjörn Ekman, and Oege De Moor. 2008. Sound and extensible renaming for Java. In *Proceedings of the Sigplan Notices*, Vol. 43. ACM, Nashville, TN, USA, 277–294.
[42] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, Lincoln, NE, USA, 201–211.
[43] Prem Kumar Singh and Aswani Kumar Ch. 2014. 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*. Springer, IIT Roorkee, India, 845–855.
[44] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the International Conference on Automated software engineering*. ACM, Antwerp, Belgium, 43–52.
[45] Anselm Strauss and Juliet Corbin. 1998. *Basics of qualitative research techniques*. Sage publications, Thousand Oaks, CA.
[46] Armstrong A Takang, Penny A Grubb, and Robert D Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.* 4, 3 (1996), 143–167.
[47] Kunal Taneja and Tao Xie. 2008. DiffGen: Automated regression unit-test generation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. IEEE, L'Aquila, Italy, 407–410.
[48] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. 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*. ACM, Amsterdam, Netherlands, 193–202.
[49] Thomas Tilley, Richard Cole, Peter Becker, and Peter Eklund. 2005. A survey of formal concept analysis support for software engineering activities. In *Formal concept analysis*. Springer, tilley2005survey, 250–271.
[50] Paolo Tonella. 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE transactions on software engineering* 29, 6 (2003), 495–509.
[51] Paolo Tonella. 2004. Formal concept analysis in software engineering. In *Proceedings of the International Conference on Software Engineering*. IEEE, Edinburgh, UK, 743–744.
[52] Jan Tretmans. 2008. Model based testing with labelled transition systems. In *Formal methods and testing*. Springer, UK, 1–38.
[53] Jianwei Wu and James Clause. 2020. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software* 168 (2020), 110639.
[54] Yiyu Yao. 2004. 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*. Springer, Uppsala, Sweden, 59–68.
[55] Benwen Zhang, Emily Hill, and James Clause. 2015. Automatically generating test templates from test names. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Lincoln, NE, USA, 506–511.
[56] Benwen Zhang, Emily Hill, and James Clause. 2016. Towards automatically generating descriptive names for unit tests. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, Singapore, Singapore, 625–636.
[57] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, San Jose, CA, USA, 385–396.

[58] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. 2019. Augmenting Java method comments generation with context information based on neural networks. *Journal of Systems and Software* 156 (2019), 328–340.