The Journal of Systems & Software 168 (2020) 110639

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A pattern-based approach to detect and improve non-descriptive test names

Jianwei Wu *, James Clause

Department of Computer and Information Sciences, University of Delaware, Newark, DE, USA

ARTICLE INFO

Article history:
Received 8 October 2019
Received in revised form 2 April 2020
Accepted 10 May 2020
Available online 15 May 2020

Keywords:
Software testing
Software quality
Documentation

ABSTRACT

Unit tests are an important artifact that supports the software development process in several ways. For example, when a test fails, its name can provide the first step towards understanding the purpose of the test. Unfortunately, unit tests often lack descriptive names. In this paper, we propose a new, pattern-based approach that can help developers improve the quality of test names of JUnit tests by making them more descriptive. It does this by detecting non-descriptive test names and in some cases, providing additional information about how the name can be improved. Our approach was assessed using an empirical evaluation on 34352 JUnit tests. The results of the evaluation show that the approach is feasible, accurate, and useful at discriminating descriptive and non-descriptive names with a 95% true-positive rate.

1. Introduction

Unit tests are an important artifact that supports the software development process in several ways. In addition to helping developers ensure the quality of their software by checking for failures (Daka and Fraser, 2014), they can also serve as an important source of documentation not only for human developers but also for automated software engineering tools (e.g., recent work on fault localization by Li et al. uses test name information Li et al., 2019). For example, when a test fails, its name can provide the first step towards understanding the purpose of the test and ultimately fixing the cause of the observed failure. Similarly, a test's name can help developers decide whether a test should be left alone, modified, or removed in response to changes in the application under test and whether the test should be included in a regression test suite.

In this work, we believe that test names are "good" if they are descriptive (i.e., they accurately summarize both the scenario and the expected outcome of the test Trenk, 2015) and "bad" if they are not descriptive. This is because descriptive names: (1) make it easier to tell if some functionality is not being tested—if a behavior is not mentioned in the name of a test, then the behavior is not being tested (2) help prevent tests that are too large or contain unrelated assertions—if a test cannot be summarized, it likely should be split into multiple tests (3) serve as documentation for the class under test—a class's supported functionality can be identified by reading the names of its tests (Zhang et al., 2015).

Unfortunately, unit tests often lack descriptive names. For example, an exploratory study by Zhang et al. found that only 9% of the 213,423 test names they considered were complete (i.e., fully described the body of test) while 62% were missing some information and 29% contained no useful information (e.g., tests named "test") (Zhang et al., 2015). Poor test names can be due to developers writing non-descriptive or incomplete names. They can also occur due to incomplete code modifications. For example, a developer may modify a test's body but fail to make the corresponding changes to the test's name. Regardless of the cause, non-descriptive test names complicate comprehension tasks and increase the costs and difficulty of software development.

Because non-descriptive names negatively impact software development, there have been several attempts to address this issue. One approach has been to automatically generate names based on implementations (e.g., Arcuri et al., 2014; Zhang et al., 2015; Daka et al., 2017). For example, Zhang et al. and Daka et al. use static and dynamic analysis, respectively, to extract important expressions from a test's body and natural language processing techniques to transform such expression into test names (Zhang et al., 2015; Daka et al., 2017). While automatically generating names from bodies eliminates the possibility of mismatches between names and bodies the generated names do not always meet with developer approval (e.g., they may not fit with existing naming conventions). Another approach is to help developers improve their existing names by suggesting improvements. For example, Høst and Østvold proposed an approach for Java methods and variables which uses a set of naming rules and related semantics (Høst and Østvold, 2009), Li et al. provided a learning-based approach to locate software faults using test name

* Corresponding author.
E-mail addresses: wjwcis@udel.edu (J. Wu), clause@udel.edu (J. Clause).

https://doi.org/10.1016/j.jss.2020.110639
0164-1212/

Table 1
Considered projects for identifying test patterns.

Project	Commit hash	# Tests
Google Guava (Google, 2018)	473f8d2	14,020
JFreeChart (Object Refinery Limited, 2018)	d03e68a	2,176
JaCoCo (Mountainminds GmbH & Co. K.G. and Contributors, 2018)	f0102f0	1,323
Weka (Waikato University, 2018)	d72b95e	436
Barbecue (Maverix, 2018)	44a8632	154
	Total	18,109

information (Li et al., 2019), and Allamanis et al. and Pradel and Sen use a model-based and a learning-based approach, respectively, to directly suggest better names or find name-based bugs to facilitate improvements (Allamanis et al., 2015; Pradel and Sen, 2018).

In this paper, we propose a new, pattern-based approach that can: (1) detect non-descriptive test names by finding mismatches between the name and body of a given JUnit test (2) provide descriptive information that consists of the main motive of test, the property to be tested in the test, and the prerequisite needed in the test or the object to be tested (see Section 2 for details) to facilitate the improvement of non-descriptive test names . Unlike existing approaches that suggesting improvements, which were designed to handle general methods, our approach is specific to JUnit tests. The narrower scope of the work allows it to take advantage of the highly repetitive structures that exist in both test names and bodies of JUnit tests (see Section 2). From a high-level point of view, the approach uses a set of predefined patterns to extract descriptive information from both a test's name and body. This information is then compared to find non-descriptive names (i.e., cases where the name does not accurately summarize the body). When a mismatch is found, the information used by the approach can help developers address the mismatch and improve the quality of the test name.

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

In particular, this work makes the following contributions:

- A novel, pattern-based approach can detect non-descriptive test names of JUnit tests and provide descriptive information about the unit tests to help developers improve existing unit tests.
- A prototype implementation of the approach as an IntelliJ IDE plugin.
- An empirical evaluation on 10 Java projects that shows: (1) the patterns are general and cover a majority of test names and bodies (2) the patterns can accurately extract descriptive information from both test names and bodies (3) the approach can accurately classify test names as either descriptive or non-descriptive .

2. Test patterns

We choose a pattern-based approach because unit tests often have similar structures that can be used to identify the purpose of a test from both its name and body. More specifically, patterns can be used to extract: (1) the **action** which is the focus of the test (i.e., what the test is testing) (2) the **predicate** which are the properties that will be checked by the test (3) the **scenario** which are the conditions under which the action is being performed or the predicate is evaluated.

As examples of the common structures shared by unit test bodies, consider the code examples shown in Fig. 1. Fig. 1(a) shows a unit test whose body consists of a `try-catch` statement. The goal of this type of test is to perform the action under an optional scenario and then to check whether the action was successful or not. In the test corpus used for pattern generation in Table 1, we found more than 2800 tests (≈14 %) shared this structure. Because of the regular structure of this type of test, it is possible to automatically extract its purpose in the form of its action, scenario, and predicate. More specifically, the action is the `method invocation` that occurs before the "fail" statement in the `try` part of the `try-catch` statement and the scenario is the object on which the action is performed. In Fig. 1(a), the action of the test is "**execute**" and the scenario of the test is "**action**", the object being tested. For another example, Fig. 1(b) presents a unit test whose body contains only a single assertion. The goal of this type of test is to compare the result of an action under a required scenario to an expected predicate. Again, this type of test is common: about 1000 tests in the corpus mentioned above (≈5 %) share this structure. For this type of test, the action and the predicate can be found by looking at the actual (second) and expected (first) arguments to the assertion statement, respectively, and the scenario will again be the object on which the action is invoked. Therefore, in Fig. 1(b), the action of the test is "**entries**", the predicate is "**getSampleElements**", and the scenario is "**multimap**".

Common patterns among test names can also be seen in the examples in Fig. 1. Fig. 1(a) shows an example where the test name (ignoring the leading "test") consists of a leading verb separated from the following noun by an underscore. In our corpus, roughly 7000 (≈35 %) test names shared this structure. For this structure, the action of the test name is the leading verb and the scenario is the following noun (i.e., in this example, the action is "Execute" and the scenario is "Action").

Similarly, Fig. 1(b) that presents a test name that consists of a single word. In our corpus, about 3400 (≈17 %) unit tests had one-word test names. In this case, the action of the test name is simply the single word contained in the name (i.e., "Entries" is the action for this example).

In the remainder of this section, we explain the process we used to identify common test name and test body patterns and present a list of the patterns that we used to detect non-descriptive names (see Section 3).

2.1. Test corpus

To identify common patterns among unit test names and bodies we considered a set of 18,109 tests comprised of the test suites from the 5 Java projects shown in Table 1. These projects are influential open-source projects taken from either Github (2018) or Slashdot Media (2018). Each project either has thousands of stars on Github (e.g., Google Guava) or has been downloaded more than 10,000 times per week on SourceForge (e.g., Weka). Each project focuses on a different domain: "Guava" is a general-purpose collection of utility classes, "JFreeChart" is a 2D chart library designed for Java applications, "JaCoCo" is a Java code coverage library often used in testing, "Weka" is a machine learning toolkit, and "Barbecue" is used for creating barcodes.

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

(a) Try-catch statement.

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

(b) Single assertion.

Fig. 1. Example test patterns.

Table 2
Test body patterns.

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

Moreover, they are written by different authors so their test suites are likely to have tests written in different ways. Due to these criteria, the patterns we identify from these projects are likely to be general, rather than specific to any one test suite from a project or author.

2.2. Test body patterns

To identify common body patterns, we used a semi-automated process based on applying frequent pattern mining to the statements contained in test bodies. We chose to operate at the statement-level for two major reasons: (1) statements are the basic syntactic component of tests and standard unit tests are composed of statements (JUnit, 2018b) (2) while the entire test serves a purpose, individual statements encapsulate sub-steps towards achieving the overall goal (Lakhotia, 1993) such as the action, scenario, and predicate .

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

The second step in the process was to apply the ClaSP frequent pattern mining algorithm (Gomariz et al., 2013) to the abstracted statements. ClaSP is a novel algorithm that utilized vertical database strategy and heuristic to mine frequent closed patterns. We chose to use ClaSP because it can efficiently mine the complete set of frequent, closed patterns from its input (Fournier-Viger et al., 2017). This means that ClaSP ensures that each mined pattern has the highest frequency among its super-patterns (i.e., closed, similar to a class and its super-classes), and that long patterns, which are relevant for our purposes, can be mined efficiently. To avoid confusion, we will refer to the output of ClaSP as "proto-patterns" as they serve as the basis for constructing our test body patterns. As the output of this step, we generated 873 proto-patterns.

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

In Fig. 2, the pattern mining process is clearly illustrated by a real-world example that is extracted during the process of pattern mining. The example is composed of three parts: (1) abstracting each statement to a number (2) using ClaSP to mine frequent patterns from the abstracted statements (3) manually examining the proto-patterns to generate test body patterns . First, we utilized an automated script to convert statements to numbers to prepare the corpus for pattern mining, and each type of statement to number pair is also stored for reference (e.g., `methodCall` to 2). Second, after the mining results of ClaSP is completely generated, we collected all generated sequences (e.g., 0 −1 7 −1 8 −1 10 −1 11 −1 3 −1) as the proto-patterns and use the statement–number pair from the first step to reconstruct those patterns (e.g., {*start*{⋆*try*{⋆*catch*{⋆}*catch*⋆}*try*⋆}*end*⋆}). Last, we performed a manual examination of the proto-patterns to generate test body patterns. From the last two lines (i.e., each of them is a reconstructed pattern with its number of matches) in Fig. 2, although both of them are mined patterns and the first one even has more matches (i.e., 1847), we only selected the second one as one kind of representation of the *Try Catch (Restricted)* body pattern shown in Fig. 6 since the first one is a spurious entry.

Because the number of proto-patterns is large, we used various grouping strategies to merge similar proto-patterns. In particular,

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

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

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

Fig. 2. Mined examples by ClaSP.

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

Fig. 3. If Else.

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

Fig. 4. Loop.

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

Fig. 5. Try Catch.

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

Fig. 6. Try Catch (Restricted).

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

Fig. 7. Try Catch (Generalized).

we found that grouping by control-flow statements was effective as such statements often define the high-level structures of test bodies. Another useful approach was to group the proto-patterns by common prefixes in order to identify statement types that were often repeated. The resulting groups of proto-patterns were further examined to eliminate ones that did not include both the special start and end of test markers and ones that did not allow for identifying the action, scenario, and predicate. Finally, the remaining proto-patterns were manually translated in to the 17 test body patterns shown in Table 2. For these selected proto-patterns, we manually examined each of them and extracted the action, predicate, and scenario from each pattern by reviewing matched test bodies from those considered projects in Table 1. In the remainder of this section each of these patterns will be described in more detail.

If Else The motive behind `If Else` body pattern is to capture a type of test body that uses an if-else condition to fulfill its task by testing a particular `method invocation` under a given object in the if part, and use the `assertions` in the else part for its evaluation. As shown in Fig. 3, the extracted action from this pattern is "⟨*action*⟩" as the first `method invocation` in the `if` part of the `if-else` statement. Then the extracted scenario under test will be the only "object" that is declared before the `if-else` statement as "⟨*scenario*⟩", and the `method invocation` positioned as the "actual" of the first `assertion` in the `else` part will be the "⟨*predicate*⟩".

Loop The motive of *Loop* body pattern is to include any test body that is trying to repetitively test a `method invocation` under a specific "object", and use its contained `assertion` to evaluate the outcomes. The action in Fig. 4 is the first `method invocation` inside the `loop` as "⟨*action*⟩", the predicate of the test is the `method invocation` - "⟨*predicate*⟩" positioned as "actual" part of the `assertion`, and the scenario of the test is the "object" used for the `loop` condition as "⟨*scenario*⟩". In addition, the while `loop`

is used here as an example, other types of `loop` are also supported.

Try Catch The motive of creating *Try Catch* body pattern is to capture many test bodies that are trying to perform a `method invocation` under a required object and then to check whether the `method invocation` was successful or not. Accordingly, the action of the body in Fig. 5 is the `method` that was invoked as "⟨*action*⟩". And the object used to invoke the `method` or the leading object declared outside the try-catch statement - "⟨*scenario*⟩" will be the scenario of the test. The `assertion` in the body is *optional*, so there might be no predicate of the test. If there is an `assertion`, then the predicate of the test is the `method invocation` - "⟨*predicate*⟩" positioned in the "actual" part of the `assertion`.

Try Catch (Restricted) The motive of *Try Catch (Restricted)* body pattern is to include a type of test body that is trying to perform a `method invocation` (i.e., action) under an optional object (i.e., scenario) and then to check if the `method invocation` was successfully performed. Accordingly, the action of the body in Fig. 6 is the method that was invoked as "⟨*action*⟩", and the object used to invoke the `method` - "⟨*scenario*⟩" will be the scenario of the test but it is *optional* for this pattern. The `assertion` is also *optional*, so there might be no predicate of the test. If there is an `assertion`, then the predicate of the test is the `method invocation` - "⟨*predicate*⟩" positioned in the "actual" part of the `assertion`. As we mentioned in Fig. 1(a), that unit test is a standard match to the "Try Catch (Restricted)".

Try Catch (Generalized) This pattern - *Try Catch (Generalized)* body pattern is a more general form of the previous two types of `try-catch` statement-based body patterns. Similarly, the motive of creating this pattern is to capture any test that is trying to perform a `method invocation` under a required object and to check if the `method invocation` was successful. The action and the scenario of the body are in the same places as mentioned in the previous two patterns - "⟨*action*⟩" and "⟨*scenario*⟩" in Fig. 7. Other statements might appear before the "⟨*scenario*⟩.⟨*action*⟩", but they are considered as "setup" for the action and scenario. The `assertion` for this pattern is still *optional*, but it could appear in the catch part or outside the `try-catch` statement. If there is an `assertion`, then the predicate of the test is the `method invocation` - "⟨*predicate*⟩" positioned in the "actual" part of the `assertion`.

All Assertion (Single) A test body matched by the *All Assertion (Single)* body pattern compares the result of an action under a required scenario to an expected predicate. In *All Assertion (Single)*, the single `assertion` contained in the test body is trying to compare different results, so the action is the `method invocation` placed in the "actual" position of the `assertion`. The predicate of the test is the `method invocation` placed in the "expected" position of the `assertion`, and the scenario will be the "object" that invokes the action. Therefore, in Fig. 8, the action of the body is "⟨*action*⟩", the predicate is "⟨*predicate*⟩" that is required for its comparison and the scenario is "⟨*scenario*⟩", which is also required to invoke the "⟨*action*⟩". Like in Fig. 1(b), the unit test is a standard match to the *All Assertion (Single)*.

All Assertion (Multiple) The *All Assertion(Multiple)* body pattern serves as a more general form of the *All Assertion (Single)* pattern. The motive and the locations of action, predicate, and scenario are the same as the *All Assertion (Single)* pattern. There are two differences: (1) the test contains more than one `assertion` as long as they are testing the same action, predicate, or scenario. (2) there is a new type of `nested method invocation` in the `assertion` . In Fig. 9, the action, predicate, and scenario for the first kind of `assertion` are the same as the `assertion` in *All Assertion (Single)*. For the second kind of `assertion`, the action of the body will be the outer `method invocation` as "⟨*action*⟩" that is invoked by the scenario of the body as "⟨*scenario*⟩". The inner `method invocation` - "⟨*predicate*⟩" is the predicate of the body, and it serves as a further step of performing the main action.

Normal (Restricted) The motive of *Normal (Restricted)* body pattern is to capture a type of test body that tries to perform an action under a specific scenario as its "setup" and evaluate it with a required predicate. The action often appears in the initialization part of the leading `declaration`, but it can also be in the "actual" part of the only `assertion`. The scenario is optional (i.e., none of the first two statements is `method invocation`), but it could be the first `method` being invoked before the final `assertion` or the `object` initialized in the leading `declaration`. The predicate is the `method` name (e.g., "assertEquals", "assertNotNull", etc.) extracted from the `assertion`. In Fig. 10, the action of the body is "⟨*action*⟩", the predicate is "⟨*predicate*⟩", and the scenario is "⟨*scenario*⟩".

Normal (Generalized) The motive of *Normal (Generalized)* body pattern is also to capture a type of test body that tries to perform an action under a specific scenario as its "setup" and evaluate it with a required predicate. This pattern is an extended version of *Normal (Restricted)*, so it shares the similar extraction of the action and predicate. One major difference between this pattern and the previous one is that this pattern allows more statements to be included in the body, and another difference is that this pattern only considers the `method invocation` as the scenario since multiple objects can be declared in the test body. In Fig. 11, the action of the body is "⟨*action*⟩", the predicate is "⟨*predicate*⟩", and the scenario is "⟨*scenario*⟩".

No Assertion From a structural perspective, the *No Assertion* body pattern differs from other patterns due to its lack of any `assertion`, which is inspired by one of the test stereotypes mentioned in a recent work (Li et al., 2018). Nonetheless, we greatly extended this type of test body as the following patterns. The motive of *No Assertion* pattern is intended to perform an action under a required scenario, but there is often no primary predicate due to the lack of `assertion`. However, this pattern requires at least three lines of code for its information extraction, and it attempts to extract the predicate of the test. The primary position of the action is in the initialization part of the leading `declaration`, and it can be as the first `method` being invoked after the `declaration`. The scenario is the object that invokes the first `method invocation` (i.e., the action) and is declared in the leading `declaration`. The predicate is the secondary `method invocation` being invoked after the first `method invocation`. In Fig. 12, the action of the body is "⟨*action*⟩", the predicate is "⟨*predicate*⟩", and the scenario is "⟨*scenario*⟩".

No Assertion (Generalized) The motive of *No Assertion (Generalized)* body pattern is also intended to perform an action under a required scenario, but there is often no primary predicate due to the lack of `assertion`. Because the required lines of code decrease to two lines, this pattern is capable of capturing more test bodies. The action changes to the first `method` being invoked after the leading `declaration`. The scenario is the `object` in the leading `declaration`. In Fig. 13, the action of is "⟨*action*⟩" and the scenario is "⟨*scenario*⟩".

No Assertion (Specialized for sole method) The *No Assertion (Specialized for sole method)* body pattern is a distinctive kind of no assertion-based pattern. The specialization of this pattern is that it only captures a test body with only one `method invocation` across all its statements, which could be an independent `method invocation` or an argument from any statement in the body. In Fig. 15, the motive of this pattern is to perform the sole action (i.e., the `method invocation`) under a required scenario, so the action of the body is "⟨*action*⟩" and the scenario of the body is "⟨*scenario*⟩".

No Assertion (Single declaration) Creating *No Assertion (Single declaration)* body pattern is to include a special kind of no assertion body pattern that can capture any test body with a sole declaration. The motive of this pattern is testing an "object" that is initialized by a required `method` to check if the "object" can be successfully initialized. In Fig. 16, the action of the body is "⟨*action*⟩" as the `method` being invoked, and the scenario of the body is "⟨*scenario*⟩" as the object being tested.

No Assertion (Single method invocation) Creating *No Assertion (Single method invocation)* body pattern is to include a special kind of no assertion body pattern that can capture any test body with a sole `method invocation`. The motive of this pattern is performing an action that is under a required argument (i.e., predicate) to check if the action can be successfully performed. In Fig. 14, the action of the body is "⟨*action*⟩" as the `method` being invoked, and the predicate of the body is the "⟨*predicate*⟩" as the inner argument of the `method invocation`.

No Assertion (Single new object) The *No Assertion (Single new object)* body pattern is also a special kind of no assertion body pattern that can capture any test body with a sole new object initialization. The motive of this pattern is initializing a `new object` that is chained to two required `methods` to check if the `new object` can be successfully initialized. In Fig. 17, the action of the body is "⟨*action*⟩" as the last `method` being invoked, the predicate is the "⟨*predicate*⟩" as the first `method` being invoked, and the scenario is "⟨*scenario*⟩" as the new object being initialized.

No Assertion (Multiple declarations) The *No Assertion (Multiple declarations)* body pattern is to create an extension of the *No Assertion (Single declaration)* pattern, and it allows more than one line of code in any captured test body. In Fig. 18, the motive and information extraction are the same as the "single method" version: the action of the body is "⟨*action*⟩" as the `method` being invoked, and the scenario is "⟨*scenario*⟩" as the object being tested. Also, the scenario in this pattern needs to be the most frequently evaluated object in the test body, and the action is served as a required argument of that object.

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

Fig. 8. All Assertion (Single).

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

Fig. 9. All assertion (Multiple).

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

Fig. 10. Normal (Restricted).

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

Fig. 11. Normal (Generalized).

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

Fig. 12. No assertion.

No Assertion (Multiple method invocations) The *No Assertion (Multiple method invocations)* body pattern is to create an extension of the *No Assertion (Single method invocation)* pattern, and it allows more than one line of code in any captured test body. The motive and the information extraction are the same as the "single method" version: the action of the body is "⟨*action*⟩" as the `method` being invoked, and the predicate is the "⟨*predicate*⟩" as the inner argument of the `method invocation`, which is also shown in Fig. 19.

Also, the action in this pattern needs to be the most frequently invoked `method` in the test body, and the predicate is associated with the action as its inner argument.

2.3. Test name patterns

Because test names are easier to compare since they are shorter than test body we were able to use a fully manual process for identifying commonalities among test names. We found that test names typically fall into two main categories: names that

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

Fig. 13. No assertion (Generalized).

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

Fig. 14. No assertion (Single method invocation).

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

Fig. 15. No assertion (Specialized for sole method).

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

Fig. 16. No assertion (Single declaration).

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

Fig. 17. No assertion (Single new object).

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

Fig. 18. No assertion (Multiple declarations).

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

Fig. 19. No assertion (Multiple method invocations).

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

Fig. 20. Divided duel verb phrase.

have a common structural format and names that have a common grammatical structure. For the first category, regular expressions can be used directly on the test names to identify relevant pieces of information. For the second category, additional information such as the part of speech of each word in the test name is needed. To obtain this information we used an approach recommended by Olney et al.: (1) convert each test name to a sentence by using a purpose-built identifier splitter and prepending the result with the word "I" (2) apply the Stanford Tagger (Stanford Natural Language Processing Group, 2018) (Olney et al., 2016). The resulting 10 name patterns are shown in Table 3 and each is described with more details in the remainder of the section.

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

Fig. 21. Is and past participle phrase.

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

Fig. 22. Verb with multiple nouns phrase.

Table 3
Test name patterns.

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

Verb With Multiple Nouns Phrase This name pattern aims to match a test name that is composed of a prefix of "test" and a verb phrase with multiple nouns. In Fig. 22, the first word after the leading "test" should be tagged as "verb" that is the action of the name, and the following three "nouns" are combined as the scenario of the name.

Divided Duel Verb Phrase The motive of this name pattern is to match a type of test names that has a leading "test" followed by a "verb–noun–verb–noun" structure. In Fig. 20, the first word tagged as "verb" is the action of the name, and the second word tagged as "noun" is the scenario of the name that is same as the fourth word. The third word should be tagged as "verb" that is the scenario of the name.

Is And Past Participle Phrase This name pattern is intended to match any test name that has a "verb–verb" structure, and the second "verb" should be in its past participle form. In Fig. 21, the first "verb" is the action of the name, and the second "verb" is the predicate of the name. If there is a following "noun" after the second "verb", it will be the scenario of the name. However, there were not enough pattern matches to support the scenario, so it is currently not included in this name pattern.

Try Catch (Name) The "Try Catch" name pattern is designed for test names, and this name pattern belongs to the regular expression-based name patterns. Fig. 23 shows a more representative sub-pattern of this pattern than other sub-patterns. In the figure, the action of the name is placed before the divider - "Throws", and the predicate of the name is placed after the divider. Moreover, this name pattern is often related to a `try-catch` condition that will be tested in the test.

Duel Verb Phrase This name pattern aims to match a type of test names that has a "verb–verb–noun" structure. In Fig. 24, the action of the name is the first word tagged as "verb", the predicate is the second word also tagged as "verb", and the scenario is the third word tagged as "noun".

Noun Phrase This name pattern is set to match any test name that only has one word tagged as a "noun". As shown in Fig. 27, the only "noun" is the scenario of the name, and there is often no action or predicate in the name.

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

Fig. 23. Try Catch (Name).

Single Entity The "Single Entity" name pattern also belongs to the regular expression-based name patterns, and a representative sub-pattern is shown in Fig. 28. After the leading "test", the combination of all following words is the action of the name. Nonetheless, the action of the name needs to fulfill a special requirement that requires the action to be matched to one of the "method under test" (Zhang et al., 2016). A "method under test" is a `method` that is being tested in the test or the test class. When the action of the name is matched to a "method under test" (i.e., identical names), it will be counted as a pattern match to this name pattern.

Verb Phrase Without Prepended Test This name pattern aims to match any test name that is a "verb phrase" without a prepended word - "test". The "verb phrase" consists of a leading "verb" with a following "noun", and there is a secondary "verb" (i.e., optional) that comes after the "noun". In Fig. 25, the action of the name is the leading "verb", the predicate of the name is the secondary "verb", and the scenario of the name is the "noun" between the action and the predicate.

Verb Phrase With Prepended Test This name pattern aims to match any test name that is a "verb phrase" with a prepended word - "test". The "verb phrase" also consists of a leading "verb" with a following "noun", and there is a secondary "verb" (i.e., optional) that comes after the "noun". In Fig. 26, the action of the name is the leading "verb", the predicate of the name is the secondary "verb", and the scenario of the name is the "noun" between the action and the predicate.

Regex Match This name pattern is a collection of 70 regular expression-based sub-patterns. For example, three of the most representative sub-patterns are shown in Fig. 29. The first two sub-patterns show a special condition that need to perform the predicate under the defined scenario or execute the action after the scenario is performed. The last sub-pattern is to execute the predicate while the right scenario is successfully performed, and a pattern match in practice is shown in Fig. 33(a).

3. A pattern-based approach to detect non-descriptive test names

Fig. 30 presents a high-level overview of our pattern-based approach for detecting non-descriptive test names. As the figure shows, the approach takes as input a unit test comprised of its name and body. It then assesses the descriptiveness of the test's name using two phases. The first phase, *pattern-based analysis*, uses the test patterns described in Section 2 to extract descriptive information from both the test name and the test body. The second phase, *information comparison*, compares the descriptive information extracted from the name and body against each other. This information comparison process allows for not only detecting non-descriptive test names (i.e., mismatches between the information), but also in some cases indicating to developers how the name could be improved. The remainder of this section describes the two steps of the approach in more detail.

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

Fig. 24. Duel verb phrase.

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

Fig. 25. Verb phrase without prepended test.

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

Fig. 26. Verb phrase with prepended test.

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

Fig. 27. Noun phrase.

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

Fig. 28. Single entity.

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

Fig. 29. Regex match.

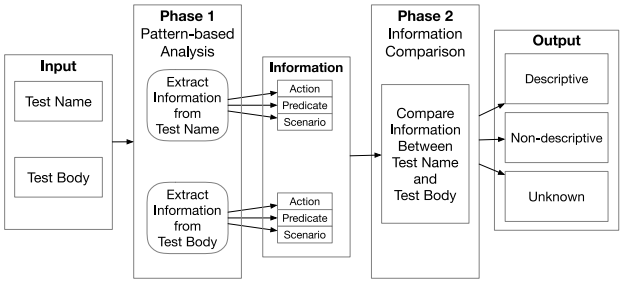

Fig. 30. Overview of the pattern-based approach.

3.1. Phase 1: Pattern-based analysis

The first phase of the approach is relatively straightforward as it consists mainly of applying the patterns described in Sections 2.2 and 2.3 to the provided test name and body. If a pattern matches against a name or body, the values it extracts as the action, predicate, and scenario are passed as input to the second stage. If none of the name patterns match or none of the body patterns match, empty values are passed instead.

Generally, if a name or body meets all requirements of a name/body pattern, the name or body is counted as a match to that name/body pattern. In Section 2, the requirements of matching each test pattern is stated in a pattern-by-pattern style. For an example of how to match the name pattern, the "Noun Phrase" name pattern can be matched to a test name that is only composed of a leading "test" and an ending noun (i.e., requirements are fulfilled, and the name is considered to be a match to the "Noun Phrase" name pattern), and the ending noun is extracted from the name as the scenario of the name. For an example of how to match the body pattern, the "All Assertion (Single)" body pattern can be matched to a test body that only contains a single and complete JUnit `assertion` (i.e., requirements are fulfilled, and the body is considered to be a match to the "All Assertion (Single)" body pattern). The `expected` part of the `assertion` is

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

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

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

(b) Comparison of information extracted by both matching name patterns for the test shown in Fig. 31a.

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

(c) Comparison of information extracted by both matching body patterns for the test shown in Fig. 31a.

Fig. 31. Example to illustrate the ordering patterns is necessary.

extracted from the body as the predicate of the body, and the `actual` part of the `assertion` should be a complete `method invocation` that contains an `object` and a `method call`. The `object` in the `method invocation` is extracted as the scenario of the body, and the `method call` is extracted as the action of the body.

After a match is found, the matched name/body pattern can extract the components from the name/body by using the corresponding positions of the action, predicate, and scenario. Similar to the two examples of matching a pattern to a test name or body that we already mentioned, the extraction process is also straightforward. For the same example of the "Noun Phrase" name pattern, the approach can automatically parse the test name with part-of-speech tags and stores every word in the name with its original order and part-of-speech tag. Then the approach first rules out irrelevant test names (e.g., test names that contain more than two words) and then extracts the first and only noun in the name to be the scenario of the name For the same example of the "All Assertion (Single)" body pattern, the approach is also able to automatically parse code from the statement-level and identifies different types of statements. Therefore, the approach first rules out any test body with more than one statement or contains any kind of statement other than JUnit `assertion`, and it then parse the JUnit `assertion` to extract the `expected` part and the `actual` part. After all parts of the `assertion` are gathered, the approach extracts the `expected` part (i.e., which is often a `method invocation`) as the predicate of the body, the `object` in the `actual` part as the scenario of the body, and the `method call` in the `actual` part as the action of the body. When every component is successfully extracted from both the test name and body, the approach will determine if the name is descriptive or non-description and generate a report for the associated test as shown Section 3.2.

The main complexity in this phase arises from the fact that more than one pattern may match a name or body. For example, Fig. 31(a) shows a unit test that can be matched by more than one pattern. More specifically, the test's body can be matched by both the restricted and generalized versions of the "Normal" pattern and the test's name can be matched by both the "Single Entity" and "Verb Phrase With Prepended Test" patterns.

While more than one pattern may match the same name or body, there is often one pattern that is preferred either because it is more accurate at extracting information or it can extract more information. For example, the difference in information extracted by matching patterns can be seen in Figs. 31(b) and 31(c). Each of these figures, the rows show the values extracted as the action, predicate, and scenario for the patterns shown in the corresponding columns. A dash (—) indicates an empty value that occurs when a pattern did not extract a value for the corresponding type of information. Fig. 31(b) is an example of when one pattern may be more accurate at extracting information. In this case, the "Single Entity" pattern correctly extracts "GetSSLProtocol" as the action and does not extract a value for the predicate or scenario while the "Verb Phrase With Prepended Test" pattern incorrectly identifies the action and scenario (i.e., "Get" is tagged as "verb" for the action and "SSL" is tagged as "noun" for the scenario). Fig. 31(c) is an example of when one pattern may extract more information. In this case, both the "Normal (Restricted)" and "Normal (Generalized)" patterns correctly identify the action as "getSslProtocol" and the predicate as "assertNotNull" but only the "Normal (Restricted)" pattern identifies the scenario as "protocol". Because of this difference in performance, it is important to order the patterns to produce the best results.

The ordering of both name and body patterns in our approach is based on our understanding of the patterns, the intuition that more specific patterns should be tried before more general patterns, and the results of applying them to the applications shown in Table 1 as a pilot study. In this pilot study, we tested ten different arrangements of the patterns and selected the one that produced the most accurate ordering. More details about this evaluation process can be find in Sections 4.2 and 4.3. The resulting orders for the name patterns and body patterns are shown in Tables 3 and 6, respectively.

3.2. Phase 2: Information comparison

The goal of the information comparison phase is to detect non-descriptive test names. Our approach fulfills this goal by comparing the information extracted from the test name and

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

(a) Example test with a descriptive name.

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

(b) Extracted information for the test shown in Fig. 32a.

Fig. 32. Example test with a descriptive name.

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

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

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

(b) Extracted information for the test shown in Fig. 33a.

Fig. 33. Examples of a non-descriptive test name.

body. The result of this comparison is that a test name is either: (1) Descriptive (2) Non-descriptive (3) Unknown .

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

If there is sufficient information to compare, the approach checks each existing piece of information from the name against the corresponding information from the body. If all of the existing pieces of information match, then the name is considered *descriptive*. Also, if all of the existing pieces from the name is a valid subset of the pieces from the body, the name is still considered *descriptive*. Fig. 32 shows an example of a test name that is classified as descriptive. The top of the figure shows the test under consideration and the bottom presents a table showing the information extracted by the first phase of the approach. The rows of the table show the values extracted by the pattern type shown in the corresponding column (i.e., the name pattern identified "GetGraphNode" as the action and the body pattern identified "getGraphNode()" as the action). In this example, the name is considered descriptive because all of the non-empty information types match their counterpart (i.e., "GetGraphNode" matches "getGraphNode()").

If, when comparing the name information against the body information, at least one of the existing pieces of information does not match, then the name is considered *non-descriptive*. It means that a subset of the following conditions happens for that name: (1) $\text{action}_{\text{name}}$ does not match $\text{action}_{\text{body}}$ (2) $\text{predicate}_{\text{name}}$ does not match $\text{predicate}_{\text{body}}$ (3) $\text{scenario}_{\text{name}}$ does not match $\text{scenario}_{\text{body}}$. Fig. 33 shows an example of name that is classified as non-descriptive. Again, the top of the figure shows the test under consideration and the bottom presents a table showing the information extracted by the first phase of the approach. In this example, the name is considered non-descriptive because none of the non-empty information types match their counterpart (i.e., "TokenIsAbsent" fails to match "response").

If the outcome is either descriptive or non-descriptive, the approach can sometimes provide additional information to developers to help them improve the test name. For both descriptive and non-descriptive names, if a value provided by the name pattern is empty but the corresponding information provided by the body is not empty, the name can likely be improved by the addition of the body information. For example, Fig. 32(b) shows a test name that is descriptive but can also be improved. In this example, the name accurately reflects that the action of the test is "GetGraphNode" but it is missing information about the predicate that can be found in the body. Adding information that the predicate is "assertEquals" to the name would improve its descriptiveness.

For only non-descriptive names, the approach can suggest modification in two cases. First, if a value provided by the name pattern exists but the corresponding value provided by the body pattern does not exist, the approach suggests that the name information from the name be removed as it is not supported by the body. Second, if the corresponding values provided by the name and body patterns both exist but do not match, the approach can suggest that the information from the name be replaced by the information from the body. For example, Fig. 33(b) shows a non-descriptive test name, which the approach can provide the following suggestions for improvement: First, the predicate part of the name, "ThrowException", should be removed and second, the scenario identified by the name, "TokenIsAbsent", should also be replaced with the scenario from the body, "response". Note that, because the action from the name is empty, the action identified by the body, "extract", should be added to the name, as described above.

The challenging part of this phase is determining whether the corresponding pieces of information match. Because the information extracted from the name is text while the information extracted from the body are code elements (i.e., `methods`, `objects`, etc.) they cannot be directly compared. To address the challenge, the approach automatically converts the *name* of any `method`, `object`, `new instance`, or `assertion method` to a string. For example, the "Normal (Restricted)" body patterns can extract the name of the `assertion method` in Fig. 31(a), and

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

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

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

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

Fig. 34. Examples of a true positive and a false positive.

it is converted to a string that is shown as "assertNotNull" in Fig. 31(c). Once both the information from the name and the information from the body have been converted to strings, they are also converted to lower case.

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

After we sorted out the process of determining a match, the pattern-based approach can automatically classify each test name in a project as a descriptive name or a non-descriptive name. In the first step, the approach gathers all unit tests from the test suite of a selected project by using an automated project analyzer and finds pattern matches for their test names and bodies. In the second step, the approach then uses the test patterns to extract the action, predicate, and scenario from the name and body of each test and generate a report for each name, which is the same as the reports shown in Fig. 34. In the last step, the approach automatically aggregates all generated reports for all extracted tests in a comprehensive report for the project, which contains all the detected descriptive and non-descriptive test names. To provide a more intuitive presentation of the approach, an implementation of the pattern-based approach is provided as an IDE plugin (Wu and Clause, 2020b).

4. Empirical evaluation

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

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

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

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

To investigate these questions, we implemented our approach as an IntelliJ IDEA Plugin (JetBrain, 2018). We chose to use IntelliJ IDEA because it is a full-featured IDE that can import projects which use a wide variety of build systems (e.g., Maven and Gradle). This gives us more flexibility in choosing applications when building our set of experimental subjects. It also has support for automatically identifying test suites, which are the input to the approach. Finally, it has a robust parsing API that we can use to implement the body patterns. Currently developers can use the plugin by selecting a menu item that analyzes all tests in the current project. In future work, the plugin could easily be extended to support other interaction mechanisms. For example, checking only the names of test in a specific class or the names of individually selected tests.

Table 4
Experimental subjects.

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

To generate the experimental data for investigating our research questions, we instrumented the plugin to record the information necessary for answering each research question. We then manually ran the plugin on each of the experimental subjects. For each run, the plugin automatically imports the project that is going to be evaluated. After the importing finishes, the plugin will attempt to match every test pattern on each unit test that is contained in that imported project. Finally, the plugin outputs a report for all unit tests that are evaluated in the process. In total, we collected all information comparison reports for each of the ten Java projects we used in the evaluation. The machine we used for all experiments was a MacBook Pro (2.7 GHz Intel i5 processor; 16 GB RAM) running macOS High Sierra and Java version 9.0.1. Adding up the time for executing the plugin on each project, the total amount of time is roughly five hours for 34,352 tests. Even though the implementation is unoptimized, the execution time is such that it is feasible to include the approach as part of an off-line build process (e.g., overnight).

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

4.1. Experimental subjects

As the subjects for the evaluation, we selected a set of 34,352 unit tests comprised of the test suites from the 10 Java projects shown in Table 4. In the table, the first column, *Project*, shows the

Table 5
Match rates for name patterns (over all tests).

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

Table 6
Match rates for body patterns (over all tests).

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

name of each project; the second column, *Commit*, shows commit hash of the version of the project that was evaluated; and the last column, *# Tests*, shows the number of unit tests contained in each project's test suite.

We chose these projects for several reasons. First, they are distinct from the applications and test suites we used to develop the patterns (see Section 2). Clearly, the patterns should perform well on the tests that they were derived from. Having separate test suites allows for a more representative evaluation of the first two research questions. Second, the applications they test are diverse since they cover a wide variety of application domains. For example, "Xodus" is a transactional database, "mytcuml" is a UML tool, "wheels" is a testing framework, and "EventBus" is a publish/subscribe pattern-based library that can simplify Android and Java code. In addition, they were written by different developers and at different times. This means that the test suites are not limited to a particular set of authors or patterns and are more likely to be representative than any test from a single project or style. Finally, in aggregate, they have a sufficient number of tests to allow for a thorough evaluation of the approach.

4.2. RQ1: Feasibility

The purpose of the first research question is to evaluate the feasibility of our pattern-based approach. The primary way in which we judge feasibility is to determine the percentage of test names and bodies that are matched by one of the patterns used by the approach. In some sense, this is the "coverage" of the patterns. If the coverage of the patterns is low, the usefulness of the approach will also be low as the approach will only be able to provide feedback for a small number of tests. Conversely, if the coverage of the patterns is high, the approach is potentially more useful as it can provide feedback for more tests. However, there is a potential trade-off between the coverage of the patterns and their accuracy (see Section 4.3) in that increasing coverage may result in lower accuracy. As such, the sweet-spot for the approach is achieving enough coverage to enable providing feedback for most tests, but not compromising the accuracy of the extracted information.

Tables 5 and 6 show the experimental data for this research question. In each table, the first column, *Name/Body Pattern*, shows the name of each pattern; the final two columns, *# Matches* and *%*, show the number of times the pattern matched against a test both as a count and a percentage, respectively; and the final row shows an overall summary of the results. For example, the fourth row of Table 5 shows that "Try Catch" matched 204 test names (i.e., ≈0.6% of the 34,352 considered test names). Similarly, the first row of Table 6 shows that "If Else" matched 17 of the 34,352 considered test bodies (i.e., ≈0.05% of the 34,352 considered test bodies). Note that, to simplify the tables and the discussion, most variations of a pattern are grouped into a single row. For example, in Table 6, "All Assertion" includes both the "Single" and the "Multiple" versions presented in Table 2.

As the final row in each table shows, the overall match rate for both name and body patterns is high. In aggregate, the name patterns matched 34,352 test names (i.e., 100%), and the body patterns matched 26,502 test bodies (i.e., ≈77%). While there are a few patterns that had low or zero match rates (e.g., "Divided Duel Verb Phrase"), the cost of keeping such patterns is low as their execution times are low and they may be more prevalent in other project types. The data also demonstrate that the ordering of the patterns is effective. More general patterns (i.e., ones have shown lower in the tables) have higher match rates than more specific patterns (ones shown higher in the tables). Overall, we believe that these results suggest that the approach is feasible. Based on the performance of several related approaches (Høst and Østvold, 2009, 2008; Zhong and Su, 2013; Singer and Kirkham, 2008; Allamanis et al., 2014), we believe that the coverage of the patterns is high enough to enable the approach to provide feedback for a majority of tests.

4.3. RQ2: Accuracy

The goal of the second research question is to investigate whether the patterns can accurately extract information from test names and bodies. Because assessing the accuracy of the extracted information must be done manually (i.e., inspect each test case with the information manually and check if it can correctly describe the action/predicate/scenario of the name or body by our researchers), it is infeasible to consider all 26,502 tests that were matched by a pattern. Therefore, we chose a subset of information extracted from matched tests to classify. For each name pattern and each body pattern, we randomly selected up to 5 tests matched by that pattern from each project. If no test was matched by that pattern in a project, we skipped the project and moved on to the next one. In total, 242 tests were selected for the name patterns and 266 tests were selected for the body patterns.

For each test in the selected subset, each author manually examined the information extracted by the matching name and body patterns independently. If the extracted information matched the human's judgment it was considered a true positive (TP) and if the extracted information did not match the human's judgment it was considered a false positive (FP). Disagreements among the raters were discussed until a resolution was reached. In total, 1524 comparisons were made by each rater (i.e., (242 tests for name patterns + 266 tests for body patterns) * 3 comparisons, the action, predicate, and scenario for each test).

Tables 7 and 8 show the experimental data for this research question. Table 7 shows the accuracy of the name patterns and Table 8 shows the accuracy of the body patterns. In each table, the first column is the name of each pattern, and the following three pairs of columns show the TP and FP rates for the information extracted as the action, predicate, and scenario by each pattern. The final row shows the overall rates for all patterns. For example,

Table 7
Accuracy results for each name pattern.

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

Table 8
Accuracy results for each body pattern.

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

Table 9
Effectiveness of the approach.

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

the fourth row of Table 7 shows the accuracy results for the "Try Catch" name pattern: the TP rate for the action is 89%, the TP rate for the predicate is 96%, and the TP rate for the scenario is 89%. Note that in Table 7 a dash (—) indicates the cases where a manual assessment was impossible because the patterns did not match any tests.

The data shown in Tables 7 and 8, indicates that the overall accuracy of both the name patterns and body patterns is high. For name patterns, the overall true positive rates range from 87% for the scenario to 92% for the action and for the body patterns the overall true positive rates range from 88% for the predicate to 97% for the scenario. Even in the worst cases (e.g., identifying the scenario with the Regex Match name pattern), the true positive rate is above 70%. As such, we believe that both types of patterns are effective at accurately identifying the action, predicate, and scenario from tests.

4.4. RQ3: Effectiveness

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

For each test in the selected subset, each author again manually examined the output of the approach independently. If the output of the approach matched the human's judgment it was considered a true positive (TP) and if the output did not match the human's judgment it was considered a false positive (FP). Disagreements among the raters were discussed until a resolution was reached. The results of the classification are shown in Table 9.

In Table 9, the first column presents each project's name, the second column shows the number of tests for each project, and the final two columns show the rates for each classification, respectively. For example, the first row shows there are 29 reports that were selected for project "Xodus", and (≈97%) of them correctly classify the test name as either descriptive or non-descriptive. The last row shows that the overall TP rate of all reports is ≈95% (i.e., 251 true positives), and the FP rate is just ≈5% (i.e., 14 false positives). Owing to the high effectiveness of our pattern-based approach, ≈99.2% of the 251 true positives are definitively correct. And the definition of correctness here is to be a suitable test name for its related test body. For instance, the test case in Fig. 32 is considered to be a true positive since the report can correctly classify its name as a descriptive test name. Additionally, more examples of true positives can be found in the public repository (Wu and Clause, 2020b). Because the true positive rate is high with nearly perfect correctness rate, we can conclude that the pattern-based approach is effective at classifying names as either descriptive or non-descriptive.

In addition, we further investigate the two examples in Fig. 34 that are selected from the 265 tests from those open-source Java projects. In Fig. 34, each example is the output that is produced by our pattern-based approach. The action, predicate, and scenario

on the left side of the equations are extracted from the test body, and the action, predicate, and scenario on the right side are extracted from the test name. For the unit test in Fig. 34(a), the test name `returnFoo2` is correctly classified as a non-descriptive test name. Also, some suggestions are provided by our approach for the example in Fig. 34(a): (1) the action of the name (i.e., `return`) should be removed from the name (2) the predicate and scenario of the name should be replaced by the predicate and scenario of the body (i.e., `equals` and `thenReturn`) . For the unit test in Fig. 34(b), the test name `shouldNotThrowSmartNullPointerOnObjectMethods` is incorrectly classified as a non-descriptive test name. Because of the difference in length between the short name and long body, the test patterns failed to correctly extract the action, predicate, and scenario from the name and body, and this example is also considered as a false positive.

5. Related work

In this paper, we propose a pattern-based approach that involves different fields of research, so the purpose of this section is to review the most closely related works that come from each field.

5.1. Detecting mismatches/improving names

There are some existing works that attempt to identify name/implementation mismatches.

Høst and Østvold's work is the most similar to our approach as it attempts to identify several types of naming bugs in general Java methods (Høst and Østvold, 2009). Their approach relies on a manually generated rule book that is extracted from the implicit convention between names and implementations in Java programming, which can be utilized to detect name bugs and provide some suggestions for constructing more suitable names. In their previous works, Høst and Østvold already showed that there is a mutual dependency between method names and their associated implementations (Høst and Østvold, 2008). Therefore, their approach considered the mismatch between the name and the implementation of its associated method and used the mismatch to fix name bugs, which are both similar to the analytical process and goal of our pattern-based approach. There are two major differences between their work and ours. First, our approach primarily focuses on the test names rather than general method names that often follow a different naming convention. For example, their approach treated the data type of the value in the `return` statement as an essential attribute in their rule book. However, normally for the unit tests, they compared different values using the `assertions` rather than any `return` statement, so the information in those `assertions` will be a crucial part of their test names. Second, instead of using a manually generated rule book, we built our approach based on the test patterns, and those test patterns were mined from a large test corpus by a semi-automatic process.

Zhong and Su provided a novel approach for detecting API documentation errors, and those errors are essentially the mismatches between the API documentation and the actual programs (Zhong and Su, 2013). To address the importance of words in Java programming, Singer and Kirkham showed that words in class names are closely related to class properties that can be described in micro patterns (Singer and Kirkham, 2008). Allamanis et al. mentioned that developers should follow a consistent naming convention, and they proposed a novel framework that can suggest identifier names accurately (Allamanis et al., 2014). All of their works comprehensively showed it is feasible to find poorly structured (i.e., non-descriptive) names by using the mismatch or pattern between the name and the program, and we can also improve those names by using providing accurate suggestions. Nonetheless, each of their techniques is often limited to a certain aspect in the problem of detecting and improving non-descriptive names, so none of them can be directly applied to improving non-descriptive names in unit tests. Pradel and Sen recently proposed a framework for the detection of naming bugs (Pradel and Sen, 2018). Regardless of their effort to introduce a new approach that can detect name-based bugs by using their machine learning method, we still cannot apply their approach to the unit tests without further modifications. Because some unit tests are expected to produce certain exceptions or failures when using them, so testers might intentionally design poorly named identifiers in those tests. Consequently, lots of false-positives could be generated without a complete retrofit to extend their proposed framework to unit tests. For instance, Junit 4 requires every test name to have a leading "test" (JUnit, 2018a), so some of existing techniques might consider the leading "test" as the action, predicate, or scenario of the name.

5.2. Automated generation of test names

In contrast to the techniques mentioned above that attempt to improve names, there are several approaches that attempt to automatically generate names.

Some of these techniques use natural language-based program analysis (NLPA). For example, Zhang et al. proposed their approach that can generate descriptive names from existing test bodies by using natural-language program analysis and text generation (Zhang et al., 2016). However, their approach left an important question unanswered that is testers need to decide which one of the three possible test names should be used for their unit tests by themselves, and it is possible that none of the three generated names follow the common naming convention. Other techniques utilized Java bytecode, method-call sequences, API-level coverage goals, and `logbilinear` context models (Fraser and Arcuri, 2011; Thummalapenta et al., 2009; Daka et al., 2017; Allamanis et al., 2015). Even with their automated generation process, their generated test names are not human-readable that can cause misunderstanding for testers who want to further modify those generated test names or bodies. Although some techniques can generate descriptive names, those techniques required testers to perform a full test execution with certain coverage goals or building a context model, which are often error-prone in practice (i.e., those coverage goals or models might be too specific to apply for certain projects).

5.3. General program analysis/automated testing and debugging

Many researchers proposed their program analysis or automated testing techniques that can help us have a better understanding of the embedded features in unit tests.

Moreno and Marcus proposed Java method and class stereotypes, and they took a closer look at the statement level analysis of Java code (Moreno and Marcus, 2012), and Ghafari et al. tried to extract the focal method under test (Ghafari et al., 2015). To be more focused on unit testing, Li et al. constructed a series of tags for distinguish unit test cases (Li et al., 2018). A group of researchers also conducted a series of works related to tagging methods or classes with stereotypes (Dragan et al., 2006, 2010; Dragan, 2011), but their works might also not be applicable for unit tests. From a general perspective of testing, other researchers tried to devise methods that can perform fully automated testing by the targeted event sequence or the environmental dependencies (Jensen et al., 2013; Arcuri et al., 2014). All of their works performed well under their specific problems in program analysis

or automated testing. However, while their works focus more on extracting features from code or automating the testing process rather than the unit tests themselves, we can still use them to improve our pattern-based approach. Furthermore, Li et al. proposed a learning-based approach for fault localization and automated debugging with high performance (Li et al., 2019), but the goal of their work is primarily to locate software faults for debugging rather than the naming of unit tests. Regardless of the performance of their proposed tool, the goal of their work is primarily to locate software faults for debugging rather than the naming of unit testing, and the experimental subjects they used is a standard benchmark database of detecting bugs rather than the unit tests from real-world Java projects.

5.4. Natural language program analysis

There are lots of existing works that try to analyze programs from a natural language-based perspective.

Pollock et al. and Shepherd et al. introduced NLPA by illustrating how to apply NLPA in practice and also giving some insights about aspect mining (Pollock et al., 2007; Shepherd et al., 2007; Pollock et al., 2009). Their studies showed natural language clues from developers' naming style can be used for improving the effectiveness of software tools. Abebe and Tonella proposed a natural language-based method to parse the identifier names of program elements for extracting concepts from them (Abebe and Tonella, 2010). Furthermore, some researchers attempted to split identifiers (Enslen et al., 2009; Butler et al., 2011; Guerrouj et al., 2013; Hill et al., 2014), and others managed to expand abbreviations (Hill et al., 2008; Madani et al., 2010; Corazza et al., 2012). Even though their works are not alternatives to our approach, we can still use their implemented tools to improve the accuracy of the test patterns.

6. Prototype implementation and threats to validity

The prototype implementation is publicly available (Wu and Clause, 2020b). All meta results from the pilot study and the pattern mining process, all the instances of non-descriptive test names from the 10 experimental subjects in Table 4, and the metadata of the evaluation are also uploaded in the repository. In addition, we are sharing data for the quantitative analysis that was performed in the evaluation (Wu and Clause, 2020a). Two threats to validity do exist for our test name/body patterns: (1) some body patterns contain a type of statement that can be cryptically constructed (2) some name patterns do not currently have a match . For the first part, although it is rare to have cryptically constructed statements in a test body, we mitigated it by supporting those cryptically constructed statements within their corresponding patterns. For example, we provided support for the conditional expression in the `If Else` body pattern to handle the cryptically constructed if else statement (e.g., `expression1?expression2:expression3;`). For the second part, we will further conduct a large-scale empirical evaluation on at least 100 Java projects in order to discover potential matches for those name patterns as part of our planned future work.

7. Conclusions and future work

Taking every test pattern into consideration, our selected test patterns can extract sufficient information from any unit test with matched name/body patterns. With the help of the output generated by our approach, developers can easily find non-descriptive test names from a given test corpus and improve those non-descriptive names by referring to the descriptive information. Furthermore, we also implemented our approach as an IntelliJ IDE plugin. In the empirical evaluation, the experimental results produced by our implemented approach are encouraging, which show our approach not only can accurately extract descriptive information from unit tests but also can correctly classify descriptive and non-descriptive test names.

For our planned future work, one possible direction is constructing an advanced version of the information comparison to improve the pattern-based approach by using more sophisticated comparing criteria. The next step should be looking into the false-positives in the evaluation to see if we can further improve existing test patterns. For example, we can further improve some name patterns by using even more accurate POS tagging or extend certain body patterns to handle different coding styles. The last step of this direction is to conduct another large-scale evaluation with at least 100 Java projects from Github as experimental subjects. To expand the scope of test patterns, an empirical study will be performed on other unit testing frameworks like `csUnit` and `PyUnit`, which are designed for `C#` and `Python`. Using the outcome of the large-scale study, we can determine if it is possible to mine and extract similar patterns from other types of unit tests and whether it might also be feasible to use mined patterns to extend the pattern-based approach to testing framework and programming languages.

CRediT authorship contribution statement

Jianwei Wu: Conceptualization, Methodology, Data curation, Software, Visualization, Writing - original draft, Writing - review & editing, Formal analysis. **James Clause:** Conceptualization, Software, Writing - review & editing, Formal analysis, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is supported in part by National Science Foundation, USA Grant No. 1527093.

References

Abebe, S.L., Tonella, P., 2010. Natural language parsing of program element names for concept extraction. In: Proceedings of the International Conference on Program Comprehension. IEEE, pp. 156–159.

Allamanis, M., Barr, E.T., Bird, C., Sutton, C., 2014. Learning natural coding conventions. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 281–293.

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

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

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2011. Improving the tokenisation of identifier names. In: Proceedings of the European Conference on Object-Oriented Programming. Springer, pp. 130–154.

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

Daka, E., Fraser, G., 2014. A survey on unit testing practices and problems. In: Proceedings of the International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 201–211.

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

Dragan, N., 2011. Emergent laws of method and class stereotypes in object oriented software. In: Proceedings of the International Conference on Software Maintenance. IEEE, pp. 550–555.
Dragan, N., Collard, M.L., Maletic, J.I., 2006. Reverse engineering method stereotypes. In: Proceedings of the International Conference on Software Maintenance. IEEE, pp. 24–34.
Dragan, N., Collard, M.L., Maletic, J.I., 2010. Automatic identification of class stereotypes. In: Proceedings of the International Conference on Software Maintenance. IEEE, pp. 1–10.
Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K., 2009. Mining source code to automatically split identifiers for software analysis. In: Proceedings of the International Working Conference on Mining Software Repositories. IEEE, pp. 71–80.
Fournier-Viger, P., Lin, J.C.-W., Kiran, R.U., Koh, Y.S., Thomas, R., 2017. A survey of sequential pattern mining. Data Sci. Pattern Recognit. 1 (1), 54–77.
Fraser, G., Arcuri, A., 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, pp. 416–419.
Ghafari, M., Ghezzi, C., Rubinov, K., 2015. Automatically identifying focal methods under test in unit test cases. In: Proceedings of the International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 61–70.
Github, 2018. Github. https://github.com/, (Accessed 20 December 2018).
Gomariz, A., Campos, M., Marin, R., Goethals, B., 2013. Clasp: An efficient algorithm for mining frequent closed sequences. In: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining. Springer, pp. 50–61.
Google, 2018. Google guava. https://github.com/google/guava, (Accessed 20 December 2018).
Guerrouj, L., Di Penta, M., Antoniol, G., Guéhéneuc, Y.-G., 2013. Tidier: An identifier splitting approach using speech recognition techniques. J. Softw.: Evol. Process 25 (6), 575–599.
Hill, E., Binkley, D., Lawrie, D., Pollock, L., Vijay-Shanker, K., 2014. An empirical study of identifier splitting techniques. Empir. Softw. Eng. 19 (6), 1754–1780.
Hill, E., Fry, Z.P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., Vijay-Shanker, K., 2008. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In: Proceedings of the International Working Conference on Mining Software Repositories. ACM, pp. 79–88.
Høst, E.W., Østvold, B.M., 2008. The java programmer's phrase book. In: Proceedings of the International Conference on Software Language Engineering. Springer, pp. 322–341.
Høst, E.W., Østvold, B.M., 2009. Debugging method names. In: Proceedings of the European Conference on Object-Oriented Programming. Springer, pp. 294–317.
Jensen, C.S., Prasad, M.R., Møller, A., 2013. Automated testing with targeted event sequence generation. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM, pp. 67–77.
JetBrain, 2018. Intellij IDEA plugin. https://www.jetbrains.com/help/idea/plugin-development-guidelines.html, (Accessed 4 April 2018).
JUnit, 2018a. JUnit 4. https://junit.org, (Accessed 20 December 2018).
JUnit, 2018b. JUnit Cookbook. https://junit.org/junit4/cookbook.html, (Accessed 20 December 2018).
Lakhotia, A., 1993. Understanding someone else's code: Analysis of experiences. J. Syst. Softw. 23 (3), 269–275.
Li, X., Li, W., Zhang, Y., Zhang, L., 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 169–180.
Li, B., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., 2018. Aiding comprehension of unit test cases and test suites with stereotype-based tagging. In: Proceedings of the International Conference on Program Comprehension. ACM, pp. 52–63.
Madani, N., Guerrouj, L., Di Penta, M., Gueheneuc, Y.-G., Antoniol, G., 2010. Recognizing words from source code identifiers using speech recognition techniques. In: Proceedings of the European Conference on Software Maintenance and Reengineering. IEEE, pp. 68–77.
Brought to you by, Maverix, S., 2018. Barbecue: Java barcode generator. https://sourceforge.net/projects/barbecue. (Accessed 20 November 2018).
Moreno, L., Marcus, A., 2012. Jstereocode: Automatically identifying method and class stereotypes in java code. In: Proceedings of the International Conference on Automated Software Engineering. IEEE/ACM, pp. 358–361.
Mountainminds GmbH & Co. K.G., Contributors, 2018. JaCoCo Java code coverage library. https://github.com/jacoco/jacoco, (Accessed 20 October 2018).
Object Refinery Limited, 2018. Jfreechart. http://www.jfree.org/jfreechart, (Accessed 20 December 2018).
Olney, W., Hill, E., Thurber, C., Lemma, B., 2016. Part of speech tagging Java method names. In: Proceedings of the International Conference on Software Maintenance and Evolution. IEEE, pp. 483–487.
Pollock, L., Vijay-Shanker, K., Hill, E., Sridhara, G., Shepherd, D., 2009. Natural language-based software analyses and tools for software maintenance. In: Software Engineering. Springer, pp. 94–125.
Pollock, L., Vijay-Shanker, K., Shepherd, D., Hill, E., Fry, Z.P., Maloor, K., 2007. Introducing natural language program analysis. In: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, pp. 15–16.
Pradel, M., Sen, K., 2018. DeepBugs: A learning approach to name-based bug detection. Proc. ACM Program. Lang. 2 (OOPSLA), 147.
Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K., 2007. Using natural language program analysis to locate and understand action-oriented concerns. In: Proceedings of the International Conference on Aspect-Oriented Software Development. ACM, pp. 212–224.
Singer, J., Kirkham, C., 2008. Exploiting the correspondence between micro patterns and class names. In: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 67–76.
Slashdot Media, 2018. SourceForge. https://sourceforge.net, (Accessed 22 December 2018).
Stanford Natural Language Processing Group, 2018. Stanford Log-linear Part-Of-Speech Tagger. https://nlp.stanford.edu/software/tagger.shtml, Accessed 4 April 2018.
Thummalapenta, S., Xie, T., Tillmann, N., De Halleux, J., Schulte, W., 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, pp. 193–202.
Trenk, A., 2015. Testing on the toilet: Writing descriptive test names. http://googletesting.blogspot.com/2014/10/testing-on-toilet-writing-descriptive.html, (Accessed 1 August 2019).
Waikato University, 2018. Weka 3: Machine learning software in Java. https://www.cs.waikato.ac.nz/ml/weka/index.html, (Accessed 20 December 2018).
Wu, J., Clause, J., 2020a. Online documents for all data of the evaluation. https://drive.google.com/drive/folders/1DsK0SpE-EeMdNBAELBzQwv8LAqT9yFHN?usp=sharing, (Accessed 27 March 2020).
Wu, J., Clause, J., 2020b. Prototype implementation. https://bitbucket.org/udse/findnamelite/src/master, (Accessed 27 March 2020).
Zhang, B., Hill, E., Clause, J., 2015. Automatically generating test templates from test names. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 506–511.
Zhang, B., Hill, E., Clause, J., 2016. Towards automatically generating descriptive names for unit tests. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 625–636.
Zhong, H., Su, Z., 2013. Detecting API documentation errors. In: Proceedings of the SIGPLAN Notices. ACM, pp. 803–816.

Jianwei Wu is a Ph.D. student at the University of Delaware. He was an undergraduate student at the Anhui University of Finance and Economics, China and finished his Bachelor of Engineering in 2016. He works as a research assistant at the University of Delaware under the advisement of Dr. James Clause. The main interests of his research are software testing and software documentation.

James Clause is an Associate Professor in the Department of Computer and Information Sciences at the University of Delaware. He received the MS and Ph.D. degrees in computer science from the University of Pittsburgh and the Georgia Institute of Technology, respectively. His primary areas of research are software testing, program analysis, green software engineering, and documentation.