Towards Automatically Generating Descriptive Names for Unit Tests

Benwen Zhang
University of Delaware
Newark, DE, USA
benwen@udel.edu

Emily Hill
Drew University
Madison, NJ, USA
emhill@drew.edu

James Clause
University of Delaware
Newark, DE, USA
clause@udel.edu

ABSTRACT

During maintenance, developers often need to understand the purpose of a test. One of the most potentially useful sources of information for understanding a test is its name. Ideally, test names are descriptive in that they accurately summarize both the scenario and the expected outcome of the test. Despite the benefits of being descriptive, test names often fall short of this goal. In this paper we present a new approach for automatically generating descriptive names for existing test bodies. Using a combination of natural-language program analysis and text generation, the technique creates names that summarize the test's scenario and the expected outcome. The results of our evaluation show that, (1) compared to alternative approaches, the names generated by our technique are significantly more similar to human-generated names and are nearly always preferred by developers, (2) the names generated by our technique are preferred over or are equivalent to the original test names in 83% of cases, and (3) our technique is several orders of magnitude faster than manually writing test names.

CCS Concepts

•Software and its engineering → Software testing and debugging; Software maintenance tools;

Keywords

Unit testing; Descriptive names; Maintenance

1. INTRODUCTION

One of the most difficult aspects of software maintenance is comprehension—understanding the software that is being modified. In fact, the amount of time needed by developers to locate and understand code is frequently greater than the amount of time that they spend making modifications [18]. In the context of testing, one of the most frequent comprehension tasks is understanding the purpose of a test. For example, when a test fails, it is necessary to understand the purpose of the test as a first step towards identifying the cause of the failure. In addition, knowing the purpose of a test is necessary to decide whether the 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.

One of the most potentially useful sources of information for understanding a test is its name. Ideally, test names are *descriptive* in that they accurately summarize both the scenario and the expected outcome of the test [30]. If a test's name is descriptive, developers no longer have to read through its body to understand its purpose. In addition, descriptive names (1) make it easier to tell if some functionality is not being tested—if a behavior is not mentioned in the name of a test, then the behavior is not being tested, (2) help prevent tests that are too large or contain unrelated assertions—if a test cannot be summarized, it likely should be split into multiple tests, and (3) serve as documentation for the class under test—a class's supported functionality can be identified by reading the names of its tests.

Despite their numerous benefits, not all tests have descriptive names. Because naming is difficult and there is no immediate downside, developers often write poor names. For example, developers may create generic test names (e.g., `test1`, `test2`, etc.) or test names that contain little information (e.g., `testAdd`, `testSubtract`, etc.). In addition, a test's name can become erroneous when it is out of sync with the test's body. For instance, a developer may modify a test's body but fail to make the corresponding changes to the test's name. Such erroneous names no longer accurately summarize the test's body. In practice, erroneous names can be more harmful than poor names. Because poor test names are often easily identifiable, developers are unlikely to consider them useful sources information. Conversely, erroneous names often appear plausible and can easily lead developers into making incorrect assumptions.

In this paper we present a new, natural language program analysis (NLPA)-based technique that can help simplify the comprehension task of understanding the purposes of tests. The technique accomplishes this by automatically generating descriptive names for unit tests. At a high-level, the technique statically analyzes tests to identify the parts of the body that correspond to the test's scenario and expected outcome. To identify these parts, the technique uses both syntactic knowledge of how unit tests are commonly structured and the semantic knowledge captured in the names of entities used by the test (e.g., variables, parameters, methods, etc.). Then, using text generation, the technique creates a

descriptive name that summarizes both the scenario and the expected outcome.

To evaluate our technique, we implemented it in a prototype tool, NameAssist, that automatically generates descriptive test names for unit tests written using the JUnit framework. Using the prototype, we conducted an empirical evaluation of the technique. The results of our evaluation are promising and show that the technique is feasible, useful, and effective.

Specifically, this work makes the following contributions:

- a novel, NLPA-based technique for automatically generating descriptive test names for unit tests
- NameAssist, a prototype implementation of the technique that automatically generates descriptive names for unit tests written using the JUnit framework
- an empirical evaluation of the technique that demonstrates that (1) compared to alternative approaches, names generated by NameAssist are significantly more similar to human-generated names and are nearly always preferred by developers, (2) names generated by NameAssist are preferred over or are judged equivalent to the original test names in 83 % of cases, and (3) the runtime costs of NameAssist are several orders of magnitude less than the amount of time needed by developers to manually generate descriptive names.

2. MOTIVATION

To motivate our technique, we performed an exploratory study of the names of 213,423 real-world test cases from over 9,000 publicly available Java projects hosted on SourceForge. Selecting such a large number of varied projects helps address the potential biases associated with small samples. Because these projects include applications of various sizes, ages, and coding styles they are likely to contain representatives of various testing styles which helps improve the generalizability of our results. To identify the test names, we initially considered all 16,183,516 methods defined in these projects. Then, based on unit testing conventions, we filtered out methods whose name does not start with "test" (except if they have an `@Test` annotation), methods that are declared inside of anonymous classes, and methods that accept parameters.

To analyze the test names, we used a custom parser for tagging test names with their parts of test (POTs) [33]. The POTs identified by the parser include the *action*, *action object*, *action modifiers*, *predicate*, and *predicate object*. In general, the action, action object, and action modifiers compose the *action phrase*, which should describe the scenario, while the predicate and predicate object compose the *predicate phrase*, which should describe the expected outcome. For example, in the test name `testSettingHeightThatIsTooSmallLeavesHeightUnchanged`, the action phrase is comprised of the action, "setting", the action object, "height", and the action modifiers, "that is too small". The predicate phrase is comprised of the predicate, "leaves unchanged", and the predicate object, "height".

Using the POTs assigned by the parser, we consider a test name to contain a full action phrase if the name has (1) an action that is some variation of "get" or "set" (e.g., "getting", "gets", etc.) and an action object, or (2) an action that is not some variation of "get" or "set" (e.g., "drawing", "lock", "send", etc.) with or without an action object. We consider a test name to contain a full predicate phrase if the name has both a predicate and a predicate object. Then, based on whether a test name contains a full action phrase and a full predicate phrase, we classify the test name as follows: *vacuous names* are test names that contain neither a full action phrase nor a full predicate phrase; *partial names* are test names that contain either a full action phrase or a full predicate phrase, but not both; and *complete names* are test names that contain both a full action phrase and a full predicate phrase.

In our set of test names, we found that ≈29 % are vacuous (62,674 out of 213,423), ≈62 % are partial (132,057 out of 213,423), and only ≈9 % are complete (18,692 out of 213,423). As this data shows, partial names are the most common. Although this may seem promising, in the majority of these cases, the test name contains only the name of the method under test. For example, we found 1,724 tests named `testEquals` and 1,087 tests named `testSerialization`. While including the name of the method under test is better than nothing, it is unlikely that such names will be sufficient to help developers understand the purposes of the tests. Vacuous names are the second most common. In general, these test names contain the word "test" optionally followed by a number. For example, we found 909 tests named `test`, 514 tests named `test1`, 342 tests named `test2`, etc. Clearly, these names are useless for understanding the purposes of the tests. Finally, complete names are the least common. This set includes test names such as `testGetSelectorThrowsClassCastException` and `testGetAdapterDoesNotAcceptNullArgument` that appear to explain both the scenario and the expected outcome.

The results of this study motivate our technique by showing that a majority of existing test names are either vacuous or partial. Because such names are not complete, they do not contain the information necessary for supporting common maintenance tasks such as understanding the purposes of tests. Our technique addresses this issue by analyzing test bodies to automatically generate descriptive names that summarize both the test's scenario and its expected outcome.

3. GENERATING DESCRIPTIVE NAMES

Figure 1 shows a high-level overview of our approach. As the figure shows, the technique requires two inputs: a test body and the application under test. It creates corresponding descriptive test names using two phases: the *analysis* phase and the *text generation* phase.

The first phase, *analysis*, involves statically analyzing the test body and application under test to identify the information that should be summarized in the descriptive name. Based on the results of our motivating study (see Section 2), we found that there are three primary pieces of information that are included in descriptive names: the action and scenario under test, which together comprise the test's scenario, and the expected outcome. The *action* is the focus of the test. It is usually an invocation of one of the class under test's instance methods but it could also be an object allocation (e.g., creating an object with `new` or a static factory method). The *expected outcome* is the part of the test that checks whether the result of performing the action matches the tester's expectation. In a unit test, this is usually one or more assertions (e.g., `assertEquals`, `assertTrue`, etc.). Finally, the *scenario under test* is the part of a test that constructs the environment in which the action should be performed. In general, the scenario under test is the largest part of the test and it frequently contains too much information to include in

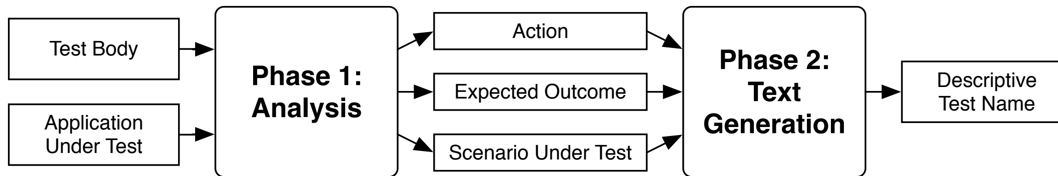

Figure 1: Overview of our approach for generating descriptive test names.

```
    public void test() {
1.    servlet = new BarcodeServlet();
2.    params.put("height", "200");
3.    params.put("width", "3");
4.    params.put("resolution", "72");
5.    req.setParameters(params);
6.    servlet.doGet(req, res);
7.    Barcode barcode = servlet.getBarcode();
8.    assertEquals(barcode.getResolution(), 72);
    }
```

(a)

small	testDoGet
medium	testDoGetResolutionIs72
full	testDoGetResolutionIs72WhenParamsResolutionIs72And SettingParameters

(b)

Figure 2: Example JUnit test case (a) and automatically generated descriptive names (b).

the test name. To prioritize the expressions included in the scenario under test, the technique uses a new data structure called the action dependency graph which takes into account both data dependencies among the elements of the scenario under test, but also semantic connections between the elements and the expected outcome.

The second phase, *text generation*, takes as input the information identified by the first phase. It then organizes the action, expected outcome, and scenario under test and translates them into a descriptive test name. Because there is a wide variety in what information testers want to include in their test's names, the text generation phase uses a template-based approach that allows testers to customize the level of information included in the descriptive names. We provide three built-in templates: *small*, which generates names that contain only the test's action; *medium*, which generates names that contain the test's action and expected outcome; and *full*, which generates names that contain the test's action, expected outcome, and scenario under test. Given a test name template, the next step is to translate the information required by the template into natural language. Because the action, expected outcome, and scenario under test are usually a small number of expression types (e.g., method invocations), we developed a rule-based approach for converting them into English phrases that follow both Java identifier restrictions and common test naming conventions. Finally, the individual phrases are concatenated together by adding necessary conjunctives and placed inside the template.

3.1 Concrete Example

As a concrete example of how our technique generates test names, consider the test shown in Figure 2a and assume that a developer wants to replace the test's existing name with a more descriptive version. For this example, the analysis phase of the technique identifies the call to `doGet` at Line 6 as the test's action; the call to `assertEquals` at Line 8 that compares the resolution of barcode to 72 as the expected outcome; and the creation of a new `BarcodeServlet` at Line 1, the calls to `put` at Lines 2-4, and the call to `setParameters` at Line 5 as the scenario under test with the calls to `setParameters` at Line 5 and `put` at Line 4 being the most important parts of the scenario under test.

Depending on the template that is used, the text generation phase generates one of the three descriptive names shown in Figure 2b. If the small template is used, the action is straightforwardly translated into the phrase "DoGet". If the medium template is used, the expected outcome is translated into "ResolutionIs72" and is appended to the small name, and if the full template is used, the most important elements of the scenario under test are translated into "ParamsResolutionIs72AndSettingParameters" and are appended to the medium name using "When" as a conjunction.

3.2 Phase 1: Analysis

The goal of the analysis phase is to identify the action, expected outcome, and scenario under test contained in a test. The remainder of this section describes how the technique uses various forms of static analysis to identify these pieces of information.

3.2.1 Identifying the Action

The first step towards identifying the action is to identify the class under test. In most cases, the class under test can be inferred by considering the name of the class that contains the test. For example, tests inside a class called `TestFoo` or `FooTest` usually test methods of class `Foo`. If this naming convention is followed, the class under test can be easily identified by stripping the leading or trailing "Test" from a test class's name. Unfortunately, identifying the class under test is not always this simple. We observed a number of counter-examples in real test suites.

To account for cases where the standard test class naming convention is not followed, the analysis phase uses a rule-based system for determining the class under test. The technique considers the following rules, in order, until it identifies the class under test: (1) if a class with the name of the test's containing class, minus "Test", exists and has one of its methods invoked by the test, it is considered the class under test, (2) if the test contains a single constructor invocation, the class whose constructor is called is considered the class under test, (3) if the test contains a single factory method and a class with the same name as the factory class, minus "Factory", exists it is considered the class under test, (4) if the test invokes a non-getter method before calling an assertion method, the non-getter method's declaring class is considered the class under test, (5) if the test invokes a method before calling an assertion method, the method's declaring class is considered the class under test, (6) if the test invokes a method as part of calling an assertion method, the method's declaring class is considered the class under test, and (7) if no other rule applies, `Object` is considered

the class under test. For example, the class containing the unit test shown in Figure 2 is called `BarcodeServletTest` that satisfies the first rule, and therefore `BarcodeServlet` is chosen as the class under test.

Identifying the class under test is a necessary prerequisite for identifying a test's action because the action is an invocation of a method declared by the class under test (or the creation of an instance of the class under test via `new` or a factory method). Because we have already identified the class under test in the first step, in the second step we only need to consider expressions that are related to the class under test as potential actions. Specifically, the potential actions include (1) invocations of methods declared by the class under test that occur before, or as part of, the assertion, and (2) instantiations of the class under test that occur before the assertion. For the example shown in Figure 2, the set of potential actions contains the instantiation of `BarcodeServlet` at Line 1, the call to `doGet` at Line 6, and the call to `getBarcode` at Line 7.

Again, we use a rule-based approach to choose the action from the set of potential actions. These rules include: (1) if the set of potential actions contains a single element (either method invocation or object instantiation), it is chosen as the action, (2) if the set of potential actions contains only one object instantiation and one getter method invocation, the object instantiation is chosen as the action, (3) if the set of potential actions contains one or more non-getter method invocations, the one closest to the assertion is chosen as the action, (4) if the set of potential actions contains one or more method invocations, the one closest to the assertion is chosen as the action, and (5) if the set of potenial actions contains one or more object instantiations, the one closest to the assertion is chosen as the action. For our running example, the third rule is the first one that is satisfied which results in the call to `doGet` at Line 6 being selected as the test's action.

Both the rules for identifying the class under test and the action were derived based on our examination of existing tests in our motivating study. While they perform well, as seen in our evaluation (Section 4), it is unlikely that they are complete. If additional experimentation would reveal shortcomings, we will adjust the approach by modifying the rule set appropriately.

3.2.2 Identifying the Expected Outcome

The expected outcome part of a test is used to check whether the result of performing the action matches the tester's expectation. In a unit test, this is done by calling a JUnit assertion method (e.g., `assertEquals`, `assertTrue`, etc.). Currently, we focus on tests with a single assertion. This choice allows us to explore the feasibility of automatically generating descriptive test names without reducing the scope of the problem too far; in our study of real tests (see Section 2), we found that a significant number follow this recommendation. In future work we plan on investigating approaches for summarizing multiple assertion that are similar to the action dependency graph (see Section 3.2.3).

While identifying calls to assertions is straightforward, generating descriptive names requires a detailed understanding not only of what method is being called but also which argument represents the tester's expectation (i.e., the expected value) and which argument represents the result of performing the action (i.e., the actual value). For single parameter

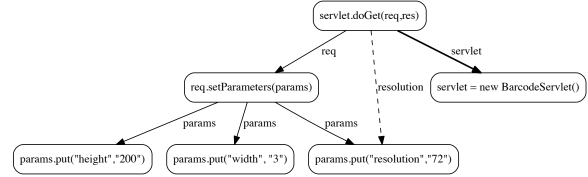

Figure 3: An example action dependency graph.

assertions (e.g., `assertTrue`, `assertFalse`, etc.), clearly the single argument is the actual value and the expected value is encoded in the assertion's name (e.g., true, false, etc.). For multiple parameter assertions (e.g., `assertEquals`), the situation is not as straightforward. Although the JUnit API declares the first parameter to be the expected value, we found that testers often switch the order of the parameters.

While reversing the argument order does not impact the test's ability to detect errors, it does mean that we cannot rely on argument order to identify the expected value. Instead we use static analysis to track backwards along each argument's use-def chains. If one argument resolves to a constant and the other to a method invocation, we consider the constant to be the expected value and the method invocation to be the actual value. If both resolve to method invocations, the technique checks whether one of the invocations calls a method declared by the class under test. If so, the other method invocation is treated as the expected value. Otherwise, the technique assumes the tester is using the API correctly and considers the first argument as the expected value. For the example shown in Figure 2a, although the order of the arguments is reversed, the technique can correctly identify 72 as the expected value since only one of the arguments is a constant. Again, these rules were defined based on our experience and appear to work well. If necessary, we will improve them in future work.

3.2.3 Identifying the Scenario Under Test

The scenario under test is the part of a test that is used to set up the necessary environment for performing the action. In order to identify it, we build an action dependency graph that is rooted at the action. Figure 3 shows the action dependency graph for our running example. At a high-level, nodes in the action dependency graph are expressions in the test and edges encode relational information among the expressions. Solid edges indicate relations among the identifiers in the source and target expressions. For example, in Figure 3, the solid edge labeled "req" indicates that the expression `req.setParameters(params)` uses the identifier "req", which is used in the action. The type of relation is indicated by the weight of the edge, with bold edges denoting that the target expression defines the identifier and normal weight edges denoting that the target expression uses the identifier. Dashed edges indicate relations among the words in the test's expected outcome and the target node. For example, the dashed edges indicate that the word "resolution" occurs both in the expected outcome, `assertEquals(72, barcode.getResolution())` and the expression `params.put("resolution", "72")`.

Capturing relations among words in addition to relations among identifiers allows the technique to identify expressions that are related to the assertion but would not otherwise be identified. It also has the benefit of reducing

the depth of expressions that are closely related to the expected outcome. For example, in Figure 3, the depth of `params.put("resolution", "72")` is reduced from two to one. This helps ensure that such expressions are included when the scenario under test is converted into English phrases (see Section 3.3).

Algorithm 1 Building the action dependency graph

Input: *Action*: identified action
Input: *Assertion*: identified expected outcome

```
1:  function buildGraph(action, assertion)
2:    g ← Graph
3:    worklist ← {action}
4:    while |worklist| ≠ 0 do
5:      current ← extract(worklist)
6:      for id ∈ Identifiers(current) do
7:        if Visited(id) then
8:          continue
9:        end if
10:       for use ∈ Uses(id) do
11:         AddUseEdge(g, current, use, id)
12:         worklist ← worklist ∪ {use}
13:       end for
14:       for def ∈ Definitions(id) do
15:         AddDefEdge(g,current, def, id)
16:         worklist ← worklist ∪ {def}
17:       end for
18:     end for
19:   end while
20:   for word ∈ Words(assertion) do
21:     for node ∈ g do
22:       if ContainsWord(node, word) then
23:         AddWordEdge(g, action, node, word)
24:       end if
25:     end for
26:   end for
27:   return g
28: end function
```

Algorithm 1 shows pseudo-code for building the action dependency graph. As input, the algorithm takes the test's identified action and expected outcome (assertion). The first part of the algorithm, Lines 3 to 19, is responsible for adding identifier-based relations. At a high-level, this part of the algorithm is a typical worklist-based iterative process. First, the worklist is initialized to contain the provided action. Then, while it is not empty, an element of the worklist is removed. Next, the algorithm iterates over each identifier in the current expression. For each use and definition of the identifier, an appropriate edge is added to the graph and the expression containing the use or definition is added to the worklist. The second part of the algorithm, Lines 20 to 26, is responsible for adding word-based relations. First, this part of the algorithm iterates over all words contained in the provided assertion. The current implementation of the Words function produces a list of words by gathering all identifiers and literal values from its argument, splitting them using a camel case-based identifier splitter, and removing stop words such as "get" and "set". Next, the algorithm iterates over each expression in the graph and determines whether the expression contains any of the words from the assertion. If so, an appropriate edge is added between the root node of the graph (action) and the expression. Currently, ContainsWord performs a simple equality check. In future work, we plan to enhance this part of the technique to detect more types of matches. For example, understanding that the number 72 and the strings "72" and "seventy two" all refer to the same concept.

3.3 Phase 2: Text Generation

The goal of the text generation phase is to generate descriptive test names using the information obtained by the analysis phase. At a high level, this phase has two steps. The first step is to select the information that will be included in the resulting test name according to the provided template. The second step is to translate the selected information into a descriptive test name.

3.3.1 Test Name Templates

Because there is a wide variety in both what information testers want to include in their test's names and how they want to present that information, the approach is configurable and allows testers to provide a template for the generated descriptive names. In this way, our technique provides testers with more options based on their preferences about test names. Both concise and very descriptive test names can be generated by modifying the templates.

Currently, the technique uses three built-in templates that were created based on common test naming patterns [33]. The *small* template only contains the test's action. The *medium template* extends the small template by also including the expected outcome. Finally, the *full* template includes the test's action, the expected outcome, and the scenario under test. In practice, names generated by the small template are similar to the partial names we encountered in our study while names generated by the medium and full templates are comparable to names that we consider to be complete.

Choosing which parts of the action and expected outcome to include is straightforward since they are both typically single expressions. However, if all of the expressions included in the scenario under test's action dependency graph were used, the resulting test names would be unacceptably long. To address this situation, the technique prioritizes the expressions in the action dependency graph based on their depth and incoming edge type. Currently, the technique selects expressions with a depth of 1 and only selects expressions with incoming definition edges if expressions with incoming use and name edges do not exist. In our running example, this results in `req.setParameters(params)` and `params.put("resolution", "72")` being selected, but not `servlet = new BarcodeServlet()`. After selecting the expressions, the technique orders them by their position in the test. While this ordering scheme is simple, we found it to be surprisingly effective. Because test names and bodies typically follow the same order, ordering by appearance results in names that closely resemble real test names.

3.3.2 Translation

The last step of our technique is to translate the selected expressions into a descriptive name. More specifically, our technique needs to translate expressions written in Java into natural language phrases and concatenate the English phrases into legal test names. In this step, our technique uses a suite of translators that are customized for various expression types. When an expression needs to be translated,

the most specific translator is used. Currently our technique implements translators for the most commonly used expressions that appear in a test body. Specifically, our technique has translators for the following expression types: (1) if the expression is a single argument assertion, it is translated into a predicate phrase using a hand-coded lookup table (e.g., `assertTrue` is translated into "is true", `assertNotNull` is translated into "is not null", etc.), (2) if the expression is a multiple argument assertion, it is translated into a linking verb or phrase (e.g., `assertEquals` is translated into "is" while `assertSame` is translated into "is identical to", etc.), (3) if the expression is an invocation of a getter method, it is translated into a noun phrase by stripping the leading "get" (e.g., `getWidth` is translated into "width"), (4) if the expression is a non-getter method invocation, it will be translated into a verb phrase using the method name (e.g., `req.setParameters(params)` is translated into "set parameters" shown in Figure 2b), (5) if the expression is a class initialization, it is translated into a gerund phrase (e.g., `new Foo()` is translated into "creating foo"), (6) if the expression is an assignment or a variable declaration, it is translated into a short english sentence (e.g., `int x = y` is translated into "x equals y"), (7) if the expression is a mathematical expression, all mathematical symbols are translated into english words or phrases (e.g., "$a > b$" is translated into "a is greater than b"), and (8) otherwise, the default translation is to simply eliminate all illegal characters for Java identifiers.

After translating the expressions into English phrases, the technique performs some final adjustments depending on where the phrase will be placed in the template. These adjustments improve the fluency of the resulting test name. For example, the expression `req.setParameters(params)` is initially translated to "setParameters". However, because this phrase is part of the scenario under test which starts with "when", we modify the verb "set" to its gerund form "setting" to improve readability. Finally the English phrases are placed in the template with multiple phrases being joined by the appropriate conjunction.

4. EVALUATION DESIGN

To evaluate our technique, we developed a prototype implementation and investigated the following questions:

RQ1: Similarity. How similar are the automatically generated test names to human generated test names?

RQ2: Human Preference. Which automatically generated test names are preferred by developers and how do the names generated by our technique compare to the original test names?

RQ3: Productivity. How much time could be saved by using our technique rather than manually writing names?

The remainder of this section discusses our prototype implementation and provides a detailed discussion of relevant data and analysis for each research question.

4.1 Prototype Tool

To experiment with our technique, we implemented it as a prototype tool, NameAssist, for automatically generating descriptive names for tests written using the JUnit testing framework. The implementation of the analysis phase is based on Eclipse's abstract syntax tree (AST) framework and implemented as an Eclipse plugin. We chose to use this framework for several reasons. First, it parses source code directly. This gives the analyzer access to variable names and other sources of semantic information that are lost after the source code is compiled. Second, Eclipse provides implementations of many types of static analysis (e.g., class hierarchy analysis, call graphs, etc.) that we can leverage. Third, Eclipse is a commonly used IDE. This allows developers to more easily access the technique and broadens the pool of subjects that we can consider in our evaluation. In order to identify relevant parts of a test body, the analyzer uses a collection of visitors to walk the corresponding AST and Eclipse's code search API to build the action dependency graph. The implementation of the text generation phase is written in Java and translates Eclipse AST expressions into natural language using the rules described in Section 3.3. The template design allows for easily extending and improving the implementation in future work.

4.2 Alternate Approaches

To conduct a suitable evaluation of NameAssist, we compared it against two alternative test name generation approaches: a random approach and a term frequency–inverse document frequency (TFIDF)-based approach. The TFIDF-based approach represents the current state-of-the-art in summarizing source code and the random approach serves as a baseline.

4.2.1 TFIDF-based Approach

TFIDF is a metric for indicating how important a term (word) is to a specific document in a corpus (collection of documents). A term's TFIDF value with respect to a document increases as a function of the term's frequency in the document, but is offset by the frequency of the term in the corpus. While TFIDF is more commonly applied to natural language, it has been successfully applied to source code [9]. Because TFIDF has been shown to be an effective method for summarizing software artifacts code [5, 9], we choose it as the basis for an alternative test name generation approach.

The TFIDF-based test name generator considers the collection of methods in a software project as a corpus and the words that appear in the project's methods as terms. To transform a software project into a corpus, we use an Eclipse plugin. First, the methods in the project are identified using Eclipse's Java model. Then terms are extracted from each method by using an AST visitor that identifies all literal values and identifiers in the method. Note that, if the method is a test, the test's name is excluded from the set of identifiers. This prevents the approach from having access to the original test names. Identifiers are then split using a custom identifier splitter based on conservative camel case splitting [11]. Finally, stop words are removed and the resulting document is added to a Lucene index.

To generate a name for a test, the approach queries the Lucene index to find the n terms with the highest TFIDF values with respect to the test's body. The words are then ordered by the position of their earliest appearance in the test and concatenated together. We chose this ordering scheme since it matches the ordering scheme used by NameAssist for the scenario under test.

4.2.2 Random Approach

To generate a name for a test, the random approach (1) identifies the set of words contained only in the body of

the test (using the same AST visitor as the TFIDF-based approach), and (2) selects n words from the set, orders them by position of earliest appearance in the test, and concatenates them together.

4.3 Considered Unit Tests

Because our evaluation involves human developers, both for generating reference test names (see Section 4.5) and evaluating generated names (see Section 4.8), we are limited in the number of tests we can consider. As a result, we chose to consider 60 tests in our evaluation, which allowed the developers to complete their tasks in a single afternoon.

To select the 60 used in our evaluation, we first generated a list of all Java projects hosted on GitHub. Second, we filtered the list to remove projects that cannot be automatically imported into Eclipse. Because all of the automated test name generation tools are Eclipse-based, selecting Eclipse-compatible projects allows us to apply the tools without modifying the projects. Third, we sorted the projects by the number of JUnit tests they contain and selected the 20 projects with the highest number of tests. Finally, we randomly selected, from the chosen projects, 60 tests that contain a single assertion.

4.4 Evaluation Measure: BLEU

At a high-level, we can view test name generation as a translation task; a descriptive test name is a translation of a test's body into natural language. By structuring the task in this way, we can use BLEU [24]—a commonly used measure for assessing the quality of text that has been translated from one language to another—to evaluate the automated test name generation techniques. Briefly, BLEU calculates a score for a candidate translation by comparing the candidate to a set of reference translations using a modified n-gram precision calculation. The calculated score ranges from 0 to 1 and indicates the similarity of the candidate to the references with scores closer to 1 denoting higher levels of similarity. In our context, a candidate translation is an automatically generated test name, the reference translations are human-generated test names, and a score of 1 indicates that an automatically generated test name is identical to at least one human-generated test name.

When calculating BLEU scores, an n-gram length greater than 1 is typically used because it takes into account order among the words, which is an important component of fluency. In our experiments, we use an n-gram length of 4 ($\text{BLEU}_{n=4}$) because experiments on natural language sentences have shown that it has the best correlation with human judgement [24]. However, because test names are not typical natural language sentences, we also calculate scores using an n-gram length of 1 ($\text{BLEU}_{n=1}$) which ignores ordering and treats the names as bags of words.

4.5 Reference Translation Creation

A test's original name is an obvious point of comparison for the automatically generated names. However, such a comparison has several issues. First, as we demonstrated in Section 2, test names are often poor. Because our goal is to generate test names that are *better* than what are typically written by developers, the original names are not necessarily the standard we want to achieve. Second, it does not take into account the possibility that multiple, equally good names may exist for the same test. Assuming that the original name is the only correct possibility is unnecessarily pessimistic. Rather we want to know if the automatically generated names are similar to *any* human generated name, not only the arbitrarily chosen original name.

To address these issues, our set of reference translations includes the original name as well as additional human generated names. To create the additional set of names, we contacted three experienced Java developers who are currently graduate students at the University of Delaware (UDel). The developers were chosen because they performed well in our software testing course and have several years of Java development experience. We gave each developer our set of 60 tests with their names removed. We then asked them to create three names, each corresponding to a common naming pattern: one containing only information about the test's action, one containing information about the test's action and predicate, and one containing information about the test's action, predicate, and any other information they felt was important to include. Note that the developers were only given brief definitions of action and predicate and examples of each, taken from real test suites. They had no knowledge of NameAssist nor how it generates names. They were also not given any guidance on word choice, phrasing, or how to structure the test names; they were free to use their best judgement and experience. Figures 4b, 4c and 4d show the various developer written names for the test shown in Figure 4a.

4.6 Name Generation Procedure

To create the automatically generated test names, we applied each test name generation technique to our set of 60 tests. For NameAssist, we generated three names, one for each of NameAssist's built-in templates. To generate names using the random and TFIDF-based approaches, we set n to be the number of words (excluding the leading "test") in the corresponding name generated by NameAssist. For example, if the name generated by NameAssist using its full template contained seven words, the corresponding full template names generated by the TFIDF-based and random approaches would also contain seven words. Figures 4b, 4c and 4d also show the test names generated by the automated approaches for the test shown in Figure 4a.

4.7 RQ1: Similarity

To investigate this research question, we compared each type of automatically generated test name with their corresponding human-generated names using their BLEU scores. For example, the NameAssist-generated name shown in Figure 4d achieved a $\text{BLEU}_{n=4}$ score of 0.67 when compared against the three developer written names shown in Figure 4d and the original name `testGetExpression_1`.

Figure 5 shows the result of this comparison in several Tukey-style box plots. The figure is faceted by the n-gram length (1 or 4) used to compute the BLEU score (rows) and the template (small, medium, or full) used to generated the test names (columns). For example, the upper left-most box plot shows the results for small test names using an n-gram length of 1. In each box plot, the x-axis shows the technique used to generate the candidate test name and the y-axis shows the BLEU score. In each box, the line and the upper and lower edges show the median and the upper and lower quartiles, respectively, and the diamond indicates the mean.

As the figure shows, scores calculated using $\text{BLEU}_{n=1}$

```
{
  NavigatorExpression navExpr = new NavigatorExpression();
  navExpr.setSpecId(NavigatorExpression.SPEC_SUPER_ID);
  navExpr.setSpecSuperLevel(4);
  String expression = navExpr.toString();
  assertEquals("super(4)", expression);
}
```

(a) `testGetExpression_1` from entando-core-engine

```
Developer  testSettingSpecSuperLevelIs4
  Written  testSetSpecSuperLevelIs4
           testToStringIsSuper4

NameAssist testToStringExpressionIsSuper4
           BLEUn=1: 1.0,  BLEUn=4: 0.51

     TFIDF testNavigatorExpressionNavExprExpressionSuper
           BLEUn=1: 0.33, BLEUn=4: 0.23

    Random testNavigatorExprIdIDEqualsSuper
           BLEUn=1: 0.17, BLEUn=4: 0.19
```

(b) Action and Predicate names

```
Developer  testSettingSpecSuperLevel
  Written  testToString
           testSetSpecSuperLevel

NameAssist testToString
           BLEUn=1: 1.0, BLEUn=4: 1.0

     TFIDF testNavExpr
           BLEUn=1: 0.0, BLEUn=4: 0.0

    Random testNavigatorLevel
           BLEUn=1: 0.5, BLEUn=4: 0.7
```

(c) Action names

```
Developer  testSettingSpecSuperLevelIs4WhenSettingSpecSuperLevelWith4
  Written  testSetSpecSuperLevelIs4WhenInitializingNavigatorExpression
           testToStringIsSuper4WhenSetSpecIdSetSpecSuperLevel

NameAssist testToStringExpressionIsSuper4WhenSettingSpecSuperLevelAndSettingSpecId
           BLEUn=1: 1.0,  BLEUn=4: 0.67

     TFIDF testNavigatorExpressionNavExprSpecSPECSUPERIDSuperLevelExpressionToEqualsSuper
           BLEUn=1: 0.54, BLEUn=4: 0.17

    Random testExpressionNavExprSpecIdSUPERIDSuperLevelStringToAssertEqualsSuper
           BLEUn=1: 0.62, BLEUn=4: 0.18
```

(d) Action, Predicate, and Other Information names

Figure 4: Human generated reference names and automatically generated names with associated BLEU scores for small (c), medium (b), and large (d) sizes.

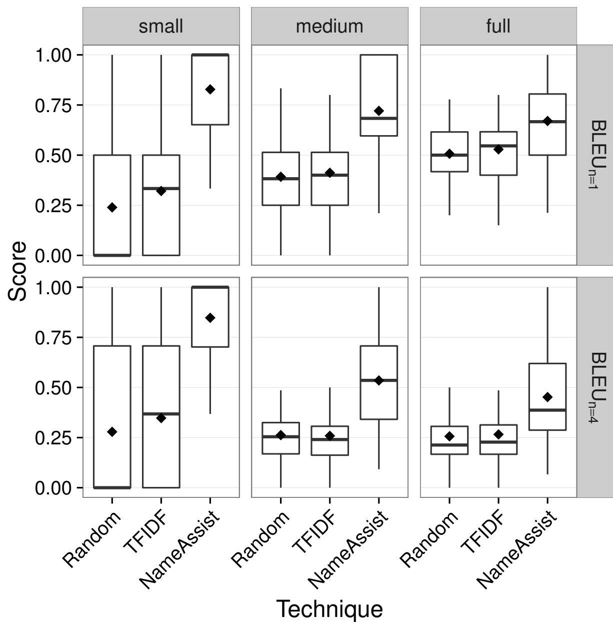

Figure 5: BLEU score distributions.

tend to be higher than scores calculated using $\text{BLEU}_{n=4}$. Intuitively, this makes sense as requiring the words in the automatically generated test names to have the same order as in the human-generated names is a more difficult task. The exception to this trend are small test names. Because small names typically only have one or two words, the ordering requirement is easier to satisfy and does not negatively impact the scores as much.

For reference, a recent comment generation tool achieves mean $\text{BLEU}_{n=4}$ scores of ≈0.54 and ≈0.63 when translating Python statements into English and Japanese pseudo-code, respectively [23]. While we cannot judge the performance of techniques in different domains by considering their BLEU scores, such comparisons can give a general sense of the relative difficulty of the translation task. Here, it appears that generating small test names is easier than generating pseudo-code while generating full test names is more difficult.

To gain more insight into the relative performance of the techniques, we performed pairwise Mann-Whitney-Wilcoxon (wilcox) tests to determine whether the populations of scores shown within each box plot are identical. We chose to use the wilcox test because we have one nominal variable (the technique used to generate the name), one measurement value (the BLEU score), and the test does not require normally distributed data (pairwise Shapiro-Wilk Normality tests indicate that it is unlikely the BLEU scores are normally distributed). The resulting p values were adjusted using Benjamini & Hochberg's false discovery rate controlling method to account for performing multiple comparisons [2]. We chose a significance level (α) of 0.05.

The results of the wilcox tests confirm our initial observations: in every case, (1) NameAssist's scores are significantly different from the scores for the TFIDF-based and random techniques (p value < 0.001), and (2) the scores for the TFIDF-based and random techniques are not significantly different from each other. Moreover, in nearly all cases, the magnitudes of NameAssist's improvements over the random and TFIDF-based techniques are "large", as measured by Cliff's Delta (i.e., $\delta > 0.474$). The only exception is for scores in the upper right boxplot (full configuration, $\text{BLEU}_{n=1}$), where the magnitudes of NameAssist's improvements are "medium" (i.e., $0.33 < \delta <= 0.474$).

Overall, we believe that these results are promising as they show that names generated by NameAssist are significantly more similar to human-generated names than the names generated by the alternative approaches.

4.8 RQ2: Human Preference

While BLEU can give a good sense of how similar a candidate translation is to a set of reference translations, it does not take into account all aspects of human judgement. For example, it cannot take into account the relative importance

Table 1: Qualitative results showing how frequently names generated by NameAssist are preferred over names generated by the alternative approaches (a) and the original test name (b).

Config.	% Majority	% NameAssist
small	98.3	96.7
medium	98.3	98.3
full	100.0	100.0

(a) Random vs. TFIDF vs. NameAssist

% Majority	% Orig.	% Equiv.	% NameAssist
98.3	16.6	11.7	71.7

(b) Original vs. NameAssist

of the words within a name (i.e., names that include many relatively unimportant words but exclude very important words may have a misleadingly high score). Consequently, we investigated our second research question by using human raters to judge (1) which of the automatically generated test names they prefer, and (2) whether they think the name generated by NameAssist is better than, equivalent to, or worse than the original name. As raters, we again selected three experienced developers who are currently graduate students at UDel. Like the developers who generated the reference names, these students were selected because they performed well in the software testing course and have several years of development experience.

To determine which of the automatically generated names the raters prefer, we showed each rater each of the 60 tests with the original name removed. Then, for each test name size, we showed the rater the three automatically generated names, in random order, and asked them to select the name they felt was the most appropriate for the test.

Table 1a shows a summary of the developers' ratings comparing the automatically generated names. The first column, *Config.* shows the test name size. The second column, *% Majority*, shows the percentage of cases where at least two of the three raters chose the same name, and the final column, *% NameAssist*, shows the percentage of cases where a majority of raters chose the name generated by NameAssist. As this data shows, in nearly all cases, the majority of raters chose the name generated by NameAssist. In fact, there were only three cases, two small names and one medium name, where the NameAssist generated name was not chosen. In two of these cases, there was no agreement among the raters (i.e., they each chose a different name) and in the final case, a majority of raters chose the name generated by the TFIDF-based approach.

To determine whether the raters think the original name is better than, equivalent to, or worse than the name generated by NameAssist, we showed each rater each of the 60 tests along with the original name and the small NameAssist generated name. Then, we asked them to them to select the name they felt was most appropriate or to indicate that the names were equivalent. Due to the threat of carryover effects (raters repeatedly seeing the original name), we were only able to compare against the original name once. Because small names are most similar to the original names in terms of information content—the mean number of words in the original names is 2.3 and the means for NameAssist's small, medium, and full names are 2.0, 7.5, and 11.4, respectively—we chose them as the most appropriate comparison point.

Table 1b shows a summary of the developers' ratings comparing original names and NameAssist names. The first column, *% Majority*, shows the percentage of cases where at least two of the three raters chose the same name. The second, third, and fourth columns, *# Orig.*, *% Equiv.*, and *% NameAssist* show the percentage of cases where a majority of raters chose the original name, felt the names were equivalent, and chose the name generated by NameAssist, respectively. As this data shows, the names generated by NameAssist compare favorably to the original test names: ≈83 % of the time, the name generated by NameAssist is preferred over or equivalent to the test's original name.

As a final step in our qualitative human evaluation, we asked the raters to informally comment, in general, on the accuracy and adequacy of the names they chose as the most appropriate (i.e., did the names focus on what the rater felt was the important aspects of the test and was any relevant information was missing from the name?). Essentially, we were interested in knowing whether the raters felt the names would be useful, or if they were simply picking the option that they considered to be the least bad. Again, their responses were encouraging. In general, they felt that the names would be useful and included the information from the tests that they felt was important to summarize.

Overall, we believe these results are also promising. They show that names generated by NameAssist are (1) nearly always preferred over names generated by the alternative techniques, (2) preferred over or are equivalent to the original test names 83 % of the time, and (3) likely to be useful in that they are not missing relevant information from the test.

4.9 RQ3: Productivity

The purpose of our third research question is to investigate how much time could be saved by using NameAssist rather having developers manually create test names. To answer this question, we compared the execution time of NameAssist against the amount of time needed by developers to manually generate names. To generate the names for the 60 tests used in our evaluation (see Section 4.5), it took each developer ≈3 h. In contrast, when executed on a modest desktop machine (3.2 GHz Intel Core i5-650 processor, 8 GB of memory, Java version 1.8.0 configured to use 4 GB of heap space), NameAssist took less than 30 s to generate the same number of names for the same tests.

Like for our other two questions, these results are encouraging. Because NameAssist's execution time is several orders of magnitude less than the amount of time needed to manually generate descriptive names, it can save significant amounts of developer time and effort.

4.10 Threats to Validity

There are a number of threats to validity of our evaluation. One threat is whether the test cases used in our evaluation are representative. To mitigate this threat, we randomly selected them from multiple open source Java programs hosted on GitHub. Because we examined many of the tests in the Sourceforge corpus from the motivating study when developing NameAssist, using a different source of tests helps control for potential overfitting and better evaluate the heuristics' applicability for new data set. Additionally, the names generated by our developers may not be representative of all

test names styles or they may share a common bias as a result of taking the same testing course. To account for this possibility, we gave them little guidance in creating the test names and we compared the automatically generated names against the original test names in addition to the developer written names. Finally, the developers in our qualitative evaluation may not have had an adequate grasp of the task or the purpose of the test. To address this threat, each developer was given a detailed document describing the task and we conducted a pilot study to detect any issues before performing the full study. In addition, we made sure to select developers with several years of professional experience.

5. RELATED WORK

In this paper, we present a new approach that combines source code summarization, program analysis, and NLPA techniques. As such, it is related to work in many different research fields. In this section, we discuss the most closely related work in each area.

5.1 Code Summarization

Code summarization techniques attempt to simplify comprehension tasks by reducing the amount of code that a developer needs to read. Within this area, Kamimura and Murphy's [17] and Li et al.'s [19] approaches for annotating tests with human-oriented summaries are the most related to our work. While their approaches also use static analysis to identify relevant parts of a test's body, their goal is to generate multi-line comments rather than descriptive names. As such, their approaches are complementary to ours; a descriptive name provides an initial overview of a test that can, if necessary, be augmented by a human-oriented summary.

Also complementary to our work are approaches for summarizing or documenting other code sequences: Sridhara et al. proposed an approach for generating summary comments for Java methods based on structural and linguistic clues [26], automatically detecting and describing high level actions within methods [27], and generating comments for method parameters [28]; Haiduc et al. investigated using various text retrieval techniques to generate summaries of both Java methods and classes; Moreno et al. proposed a technique to generate natural language summaries for Java classes by using code stereotypes and other heuristics [22]; Ying and Robillard presented a machine learning based technique for generating summaries of code fragments on the web pages [32]; McBurney and McMillan combine source code summarization and contextual information, gathered via static program analysis, to generate documentation [21]; and Allamanis et al. propose a neural probabilistic language model for source code that is used to generate variable, method and class names [1]. While Allamanis et al.'s approach could be applied to generate names for tests, they also recommend building the language model using a training set with high quality names that are rarely existing in most projects' test suites as we discussed in the motivation study. Because the focus of these techniques is generating summaries or names for other types of code or situations, they are unlikely to generate descriptive test names without significant modification.

5.2 Name/Implementation Mismatches

Within the area of name/implementation mismatches, researchers have investigated techniques for detecting API documentation errors. For example, Zhong and Su propose an approach that achieves this goal using a combination of NLPA and program analysis [34]. In addition to documentation errors, researchers have also investigated techniques for detecting, and suggesting fixes for, method name bugs— situations where a method's name does not match its implementation: Høst and Østvold use method naming conventions, mined from 100 Java projects, to identify elements (loops, method calls, field accesses, etc.) that should be present in a method's body [14]; and Suzuki et al. use an n-gram model to detect method names that are incomprehensible and to suggest the next word that should appear in a method name, given a prefix . While Høst and Østvold's and Suzuki et al.'s approaches could be applied to tests, they are ill-suited to this purpose. The structural clues that they use, such as non-void return types, parameter types, and loops, are not typically present in tests.

5.3 Natural Language Program Analysis

Finally, in the area of NLPA, researchers have investigated identifier splitting techniques (e.g., [3, 6, 7, 11]); abbreviation expansion techniques (e.g., [4, 10, 20]); techniques for tagging words with their part of speech and identifying large semantic structures (e.g., [8]); techniques for identifying programming-specific synonyms and antonyms (e.g., [12, 16, 25, 31]); and investigations into the lexicons used by programmers (e.g., [13, 15]. Again, these techniques are not alternatives to our approach but can be used to improve its accuracy and effectiveness.

6. CONCLUSIONS AND FUTURE WORK

We presented a novel NLPA-based technique for automatically generating descriptive names for existing tests that contain a single assertion. The technique helps reduce maintenance costs by simplifying the comprehension task of understanding the purpose of a test. We also presented NameAssist, a prototype implementation of the technique that generates descriptive names for unit tests written using the JUnit framework. We evaluated NameAssist to provide initial evidence that the technique is feasible and useful: (1) the BLEU scores of names generated by NameAssist are significantly higher than the scores of names generated by the TFIDF-based and random approaches, (2) names generated by NameAssist are nearly always preferred over names generated by the alternative approaches, (3) names generated by NameAssist are preferred over or are judged equivalent to the original test names 83 % of the time, and (4) the runtime costs of NameAssist are several orders of magnitude less than the amount of time needed by developers to manually generate descriptive names.

In addition to extending the evaluation, we see several directions for future work. First, dynamic program analysis could be an alternative technique for identifying the information of a test. Second, although it is difficult to summarize and name large and complex test bodies with multiple assertions, our technique may be modified to help test developers split a large test into several smaller tests with appropriate names.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 38–49, 2015.

[2] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[3] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the tokenisation of identifier names. In *Proceedings of the 25th European conference on Object-oriented programming*, pages 130–154, 2011.

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

[5] B. Eddy, J. Robinson, N. Kraft, and J. Carver. Evaluating source code summarization techniques: Replication and expansion. In *Proceedings of the 2013 IEEE 21st International Conference on Program Comprehension*, pages 13–22, 2013.

[6] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, pages 71–80, 2009.

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

[8] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 3–12, 2013.

[9] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 35–44, 2010.

[10] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 5th International Working Conference on Mining Software Repositories*, pages 79–88, 2008.

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

[12] E. Høst and B. Østvold. Canonical method names for Java. In *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 226–245. 2011.

[13] E. W. Høst and B. M. Østvold. The programmer's lexicon, volume I: The verbs. In *SCAM '07: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 193–202, 2007.

[14] E. W. Høst and B. M. Østvold. Debugging method names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 294–317, 2009.

[15] E. W. Høst and B. M. Østvold. The Java programmer's phrase book. In *Proceedings of the 1st International Conference on Software Language Engineering*, pages 322–341, 2009.

[16] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 377–386, 2013.

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

[18] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32 (12):971–987, 2006.

[19] B. Li, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and N. Kraft. Automatically documenting unit test cases. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 341–352, 2016.

[20] N. Madani, L. Guerrouj, M. Di Penta, Y. Gueheneuc, and G. Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 68–77, 2010.

[21] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290, 2014.

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

[23] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *30th IEEE/ACM International Conference on Automated Software Engineering*, pages 574–584, 2015.

[24] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 311–318, 2002.

[25] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 123–132, 2008.

[26] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52, 2010.

[27] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering*, pages 101–110, 2011.

[28] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, pages 71–80, 2011.

[29] T. Suzuki, K. Sakamoto, F. Ishikawa, and S. Honiden. An approach for evaluating and suggesting method names using n-gram models. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 271–274, 2014.

[30] A. Trenk. Testing on the toilet: Writing descriptive test names. http://googletesting.blogspot.com/2014/10/testing-on-toilet-writing-descriptive.html, 2015.

[31] J. Yang and L. Tan. Inferring semantically related words from software context. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12)*, pages 161–170, 2012.

[32] A. T. T. Ying and M. P. Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 655–658, 2013.

[33] B. Zhang, E. Hill, and J. Clause. Automatically generating test templates from test names. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 506–511, 2015.

[34] H. Zhong and Z. Su. Detecting API documentation errors. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 803–816, 2013.