Generating Unit Tests with Descriptive Names or: Would you name your children thing1 and thing2?

...
...
...

ABSTRACT

The name of a unit test helps developers to understand the purpose and scenario of the test, and test names support developers when navigating amongst sets of unit tests. When unit tests are generated automatically, however, they tend to be given non-descriptive names such as "test0", which provide none of the benefits a descriptive name can give a test. The underlying challenge is that automatically generated tests typically do not represent real scenarios and have no clear purpose other than covering code, which makes naming them difficult. In this paper, we present an automated approach which generates descriptive names for automatically generated unit tests by summarizing API-level coverage goals. The tests are optimized to be short, descriptive of the test, have a clear relation to the covered code under test, and allow developers to uniquely distinguish tests in a test suite. An empirical evaluation with 47 participants shows that developers agree with the synthesized names, and the synthesized names are equally descriptive as manually written names. Study participants were even more accurate and faster at matching code and tests with synthesized names compared to manually derived names.

ACM Reference format:
.... 2016. Generating Unit Tests with Descriptive Names
or: Would you name your children thing1 and thing2?. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software developers frequently interact with unit tests: When trying to understand code, unit tests can be consulted as usage examples. When maintaining code, unit tests help to identify undesired side-effects. When changing code, unit tests need to be updated to reflect the changed behavior. Providing tests with good names simplifies all these tasks, which is important considering the substantial costs and effort of software maintenance [7].

For example, consider the artificial example class `ShoppingCart` (Figure 1), which has two methods `addPrice` and `getTotal`. Given a test named `addPriceThrowsIllegalArgumentException` we can immediately see, even without using the test's source code, what the purpose of the test is (call `addPrice` with an argument that makes it throw an `IllegalArgumentException`), which part of the code it uses (method `addPrice`), and it is reasonable to assume that the test provides an example of unintended usage of the class `ShoppingCart`. Tests named `getTotalReturningZero` and `getTotalReturningPositive` would immediately reveal with their name that they provide two different scenarios for the `getTotal` method. When modifying the `getTotal` method, a developer would know that these tests are the first ones to run, and when one of these tests fails during continuous integration, the developer would know immediately where to start debugging.

Unit tests can be generated automatically to save time and effort, or to improve the code coverage achieved by manually written tests. Although automated test generation tools can produce tests that achieve high code coverage, these tests typically come without meaningful names. For example, the EvoSuite [13] and Randoop [32] tools name their tests "test0", "test1". These names give no hint on the content of the tests, and navigating such tests by name is impossible. Thus, even though the tests might achieve good code coverage, there is reason for concern when it comes to understanding, debugging, and maintaining these tests. The challenge, however, is that automatically generated tests tend to be non-sensical and have no clear purpose other than covering code, which makes it difficult to apply standard conventions to derive good names. Indeed, when the only purpose of a generated unit test is to cover line 8 of a class, then naively capturing this with a name like "testCoversLine27" is not helpful either.

To overcome this problem, in this paper we propose a novel technique to generate descriptive names for automatically generated unit tests. This technique is based on the insight that, while an individual generated test might not have a clearly discernible purpose on its own, the context of the test suite it is embedded in provides sufficient information to derive names which (a) describe the test's code, (b) uniquely identify the test within its test suite, and (c) provide a direct link from source code to test name. Our technique first identifies all possible descriptive elements that are identifiable at the level of the test code, then selects a minimal set of these elements for each test in a test set, and finally uses this minimal set to synthesize a descriptive, unique name for each test.

In detail, the contributions of this paper are as follows:

- A technique to synthesize descriptive names for generated unit tests in terms of their observable behavior at the level of test code.
- An open source implementation of the test naming technique, as an extension to the open source EvoSuite test generation tool [13].
- An empirical evaluation comparing manually derived names with automatically generated names in terms of developer agreement, ability to identify tests by their names, and ability to identify tests based on the code under test.

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

```
public class ShoppingCart {
  private int total = 0;
  private final static int MAX = 1000;

  public boolean addPrice(int cost) throws IllegalArgumentException {
     if (cost <= 0)
       throw new IllegalArgumentException("Cost cannot be negative");

     if (cost < MAX) {
       total += cost;
       return true;
     } else {
       return false;
     }
  }
  public int getTotal() {
     return total;
  }
}
```

Figure 1: Example class: A shopping cart that keeps track of the money spent, but has a limit for individual purchases.

In our study, participants agreed with the names synthesized with our technique. The synthesized names are as descriptive as manually written names, participants were slightly better at matching tests with synthesized names than they were at matching tests with manually written names. Finally, participants of our study were more accurate at identifying relevant tests for given pieces of code using synthesized test names.

2 BACKGROUND

Unit testing is a common activity during development of object oriented software. In this context, a unit test is a small, simple piece of code that exercises an aspect of functionality of a class under test, and checks the resulting behavior. It is common practice to have large sets of such unit tests which are executed frequently, and it is possible to generate some of the unit tests automatically. In this section we provide an overview of automated test generation and the wider problem of naming unit tests.

2.1 Unit test generation

To improve their test suites, developers can choose to use automated test generation tools. A common approach is to generate random sequences of calls [5, 11, 28, 30, 32, 35]. These sequences either serve to reveal faults such as undeclared exceptions [11] and violations of code contracts [32], or they can be enhanced with regression assertions [17, 43] which capture the current state, for example by asserting the value of observed return value. These tests can then be integrated into the code base and are used as standard regression tests, such that they will be re-executed frequently to check whether modifications on the code lead to undesired side-effects.

Because random test generation tends to result in large sets of potentially long test cases, there are approaches to minimize tests [26], or to use search-based techniques in order to generate the smallest possible test suites that achieve the highest possible code coverage [6, 14, 41]. Many optimisations focus on increasing the code coverage, for example by integrating dynamic symbolic execution [18, 21, 39] or parameterized unit tests [40]. Specifically for the Java programming language, there are also several commercial tools such as Agitar [8] and JTest [22].

Table 1: Unit tests and their names, generated for the example Java class (Figure 1).

Tool	Ref.	Test Names
AgitarOne	[2]	`testConstructor`, `testGetTotal`, `testAddPrice`, `testAddPrice1`
CodePro	[10]	`testShoppingCart_1`, `testGetTotal_1`, `testAddPrice_1`, `testAddPrice_2`, `testAddPrice_3`
eToc	[41]	`testCase1`, `testCase2`
EvoSuite	[13]	`test0`, `test1`, `test2`, `test3`, `test4`
GRT	[29]	`test01`, `test02`, ..., `test15`
JCrasher	[11]	`test0`, `test1`, `test2`
JTest	[22]	`testAddPrice0`, `testAddPrice1`, `testAddPrice2`, `testGetTotal0`
JTExpert	[37]	`TestCase0`, `TestCase1`, `TestCase2`
Randoop	[32]	`test01`, `test02`, ..., `test21`

One commonality of most of the exiting tools is that they do not generate descriptive names. Table 1 lists the names of the tests generated by various available Java test generation tools. Commercial tools, like AgitarOne, CodePro Analytix, or JTest, seem to be based on systematic algorithms that generate individual tests per method of the CUT, and thus the name is built using the name of the method under test. A minor difference lies in how constructors are handled; AgitarOne calls constructor tests `testConstructor`, whereas CodePro Analytix calls tests `test`*ClassName*. All other tools are based on random or search-based generation, and use a simple naming scheme based on a prefix (e.g., `test` or `TestCase`) plus a numeric suffix. The main commonality of all these names is that they are not very helpful to developers.

2.2 How humans name their tests

Zhang et al. [45] found that 29% of the test names in a corpus of 213,423 manually written tests were constructed using the word "test", optionally followed by a number. However, the majority of tests have at least somewhat more descriptive names. There is no consensus on what an ideal test name is, but there are common recommended guidelines and conventions.

A common approach is to name tests after the method they exercise; according to Zhang et al., 62% of names include the name of the method under test. If the method under test is simple and only has a single behavior, then naming a test by the method it calls may be sufficient (e.g., `testGetTotal`). However, often methods are not as simple. A widely used strategy [31, 42] requires three parts in a good unit test name: the method under test, the state under test, and the expected behavior; all three parts should be included in a single name, although the order of expected output and state under test can vary. For example, a test for the `ShoppingCart` example class in which the `IllegalArgumentException` is triggered (expected output) when the `addPrice` method is called with negative input (state under test) could be named:

```
addPrice_WithNegative_ThrowsIllegalArgumentException
```

Whether or not underscore characters should be used in a test name is a controversial question without clear consensus. Generally, naming guidelines and conventions expect good names to encode a fair amount of detail and to explain the specific scenario under test.

2.3 Automatically generating test names

As finding descriptive names is difficult, there has been work on generating names automatically. Host and Ostvold [19] mine naming rules from a corpus of source code, and then suggest new names for methods that do not match these rules. More recently, Allamanis et al. [3] applied a log-bilinear neural network to learn a model that can suggest method names based on features extracted from the source code. When it comes to unit tests, however, many of the features that such approaches exploit (e.g., structure of the source code) are not available, as tests tend to be simple, short sequences of calls. In particular, for automatically generated tests other important features are also absent, such as descriptive variable names.

Since generating names for tests is a somewhat different problem than generating generic method names, Zhang et al. [45] proposed an approach to generate names for unit tests based on the common structure of tests and their names. In particular, for a given test they identify the action (e.g., the method under test), the scenario under test (e.g., the parameters and context of the action), and the expected outcome (e.g., the assertion). These three parts are then converted to text using a template-based approach, which includes only the action, the action and the expected outcome, or the action, the expected outcome and a summary of the scenario under test. Even if the question of which of the three pattern types should be used in practice is answered, automatically generated tests are problematic again: The description of the scenario under test relies on descriptive variable names, which generated tests generally do not have. The expected outcome is assumed to be a single test assertion, but generated tests often have many assertions. Finally, since automatically generated tests are often generated for coverage, they may target more than one method under test.

2.4 Alternative approaches to improve test readability / maintenance effort

The ultimate aim of providing unit tests with good names is to improve understandability, and thus to improve software maintenance activities that involve unit tests. There are, however, alternative approaches to aim to achieve the same. Natural language test summaries [34] or test documentation [27] can help to make tests understandable [34]. Tests can be made easier to understand by simplifying them, for example by reducing the number of statements [25, 26], by reducing the number of assertions [17], or by splitting tests to one for each assertion [44]. Search-based approaches make it possible to include additional optimisation goals aiming to make tests more understandable, for example by making them more similar to realistic object usage scenarios [16], by making generated strings resemble natural language text [1], by optimising the syntactic readability [12], or by improving coupling and cohesion [33]. One commonality of all these approaches is that they are orthogonal to the problem of finding a good method name.

3 NAMING TESTS BASED ON COVERAGE

The main challenge in naming automatically generated tests is that these tests usually lack a real scenario or purpose that could be used to derive meaningful names. However, it is still possible to generate names that provide some of the benefits good names for manually written tests also offer. We identify the following three requirements for good names for generated tests:

R1 Test names should be descriptive of the test code; there should be an understandable, intuitive relation between the test code and its name.

R2 Test names should uniquely distinguish tests within a test suite, such that developers can use them to navigate the test suite.

R3 Test names should have a clear relation to the code under test; they should allow developers to identify relevant tests without having to see the code of the test.

In this section, we describe a naming approach tailored to a scenario where tests are generated for code coverage — perhaps the most common test generation scenario. In general, a code coverage criterion can be represented as a finite set of distinct goals, such that a test suite is considered adequate if for each goal there exists at least one test that covers it. The proposed approach uses coverage goals as the basis for name generation. While coverage goals do not describe a real test intent, they serve as a reasonable approximation since they can describe what the test *does*.

At a high level, the name synthesis first uses dynamic analysis to determine, for a selection of relevant criteria, the set of covered goals for each test in a test suite. Then, a name is constructed by selecting a minimal subset of the covered goals that is sufficient to create a unique name for each test, such that no other test in the same test suite has the same name.

Table 2: Input/output coverage goals for different datatypes

Datatype	Goals
Boolean	`True`, `False`
Numeric	`Negative`, `Zero`, `Positive`
Character	`Alphabetic`, `Digit`, `OtherChar`
Array	`Null`, `EmptyArray`, `NonEmptyArray`
List	`Null`, `EmptyList`, `NonEmptyList`
Set	`Null`, `EmptySet`, `NonEmptySet`
Map	`Null`, `EmptyMap`, `NonEmptyMap`
String	`Null`, `EmptyString`, `NonEmptyString`
Object	`Null`, *Goals derived from observers*

3.1 Coverage criteria for test naming

Not all coverage criteria are suitable for generating names. For example, assume a test that covers lines 27 to 36 of a class under test (CUT), which could be named `testLine27to36`. While the name satisfies the link between code under test and name (R3) and very likely is unique in the context of a test suite (R2), the name is hardly descriptive of the test code (R1). To satisfy this requirement, coverage goals need to be based on the *observable* behavior. That is, any values, calls, exceptions that exist in the scope of the test code, represent information that a developer could understand even without having to consult the code under test. Conceptually, we can think of these aspects as high-level coverage criteria. For each such high-level coverage criterion, we define a pattern to convert coverage goals into text, and the overall test name is a result of combinations of such names. In particular, we consider the following criteria, which are sometimes used by search-based test generation tools when generating tests [36]:

Method coverage. The most general criterion is given by the methods of the class under test; for each public method or constructor, there is one coverage goal. For a method coverage goal, the text representation simply is the method name. In the case of constructor calls, there is no method name, therefore we use the name of the class to generate a name. For example, a coverage goals for a constructor generating an instance of `ShoppingCart` would result in the text `CreatesShoppingCart`.

Exception coverage. Exceptions are generally treated special in unit tests: First, the source code structure is typically different (using a try/catch statement), and second, they often represent the main target behavior of a test. Exception coverage defines a coverage goal for each declared exception of every method of a class under test. In addition, any undeclared exceptions arising during test generation are also typically retained in a generated test suite, and can thus be counted as coverage goals. An exception coverage goal is converted to a string simply by concatenating the method name, the keyword `Throws`, and the name of the exception class (without package name). For example, method `addPrice` of class `ShoppingCart` can throw an `IllegalArgumentException`, and this would be represented as `GetCodeThrowsIllegalArgumentException`. For constructors, we use the classname again; for example, if a constructor throws an exception while trying to instantiate the `ShoppingCart` class, this coverage goal would result in the name `FailsToCreateShoppingCartThrowsIllegalArgumentException`.

Output coverage. The concept of output coverage [4] is based on the intuition that uniquely different output values are suitable as test adequacy criteria. Table 2 summarizes different output partitions for different possible return values of tested methods. For objects, we include output goals based on observers, i.e., methods that (1) do not change the object state, (2) take no parameters, and (3) return a primitive (numeric, character, string) type. For each observer, we define output coverage goals as listed in Table 2. For example, if a return value is a `Set`, then the observer `isEmpty` would result in two coverage goals (true, false), and the observer `size` would result in three output goals (<0, 0, >0). Names for output coverage goals are created by concatenating the name of the method with the keyword `Returning`, followed by the descriptor of the actual return value (as shown in Table 2). For example, a coverage goal that represents `getTotal` returning 0 would be named `GetTotalReturningZero`. When coverage goals are based on observers, then the name consists the method name, the keyword Where, then the name of the observer, followed by the keyword `Is`, and a textual description of the return value of the observer. For example: `testFooWhereIsEmptyIsTrue`.

Input coverage. The same partitioning of value ranges described above for outputs can also be applied to all input parameters. Names for input coverage goals are created by concatenating the method name with the keyword `With`, followed by a descriptor of the parameter value. For example: `testAddPriceWithZero`.

Note that, in a unit testing scenario, we assume there is a dedicated CUT, and hence only method calls on instances of the CUT, exceptions thrown in calls to the CUT, as well as inputs and outputs of CUT methods are considered for coverage goals.

Table 3: Coverage goal list for the `ShoppingCart` class, and their string representation

	Goals in class `ShoppingCart`	Text representation
Method	`ShoppingCart()`	`createsShoppingCart`
	`int getTotal()`	`getTotal`
	`boolean addPrice(int)`	`addPrice`
Exc.	`boolean addPrice(int)` → `java.lang.IllegalArgumentException`	`addPriceThrows IllegalArgumentException`
Output	`int getTotal()` → `== 0`	`getTotalReturningZero`
	`int getTotal()` → `< 0`	`getTotalReturningNegative`
	`int getTotal()` → `> 0`	`getTotalReturningPositive`
	`boolean addPrice(int)` → `False`	`addPriceReturningFalse`
	`boolean addPrice(int)` → `True`	`addPriceReturningTrue`
Input	`boolean addPrice(int)` → `== 0`	`addPriceWithZero`
	`boolean addPrice(int)` → `> 0`	`addPriceWithPositive`
	`boolean addPrice(int)` → `< 0`	`addPriceWithNegative`

Table 4: Method names for overloaded methods

Method signature	Method name
`void foo(int x)`	`fooTakingInt`
`void foo(int x, int y)`	`fooTakingTwoInts`
`void foo(int x, String z)`	`fooTakingIntAndString`
`void foo(int x,int y,int z)`	`fooTaking3Arguments`
`Foo()`	`CreatesFooTakingNoArguments`
`Foo(int x)`	`CreatesFooTakingInt`

3.2 Finding unique method names

Method names are part of all criteria listed above. However, if methods are overloaded, then method names are not unique, and thus not sufficient to uniquely identify which method a name refers to. Consequently, we include aspects of the method signature to generate a unique method name. If the number of arguments is sufficient to uniquely identify a method, we include this in the name. Table 4 shows an overloaded method `foo` with five different variants. If there is no argument, then the suffix `TakingNoArgument` is appended to the method name. If there is exactly one argument, then its type name is appended to the method name. If there are two or more parameters but no other method with the same name and same number of arguments, then `TakingXArguments` represents the name. Finally, if there are several methods with the same name and the same number of arguments, then the name is constructed from the actual signature. To abbreviate names, for each type we use the number of arguments that have this type to construct the name (e.g., `fooTakingTwoInts`). If there are overloaded constructors, then the same principle applies, except that the argument descriptor string is appended to the `CreatesFoo` string.

3.3 Synthesising coverage-based test names

The overall technique to synthesize test names is summarized in Algorithm 1. Given a test and a set of coverage criteria, represented by their coverage goals, a basic dynamic analysis (labelled CoveredGoals in Algorithm 1) can determine which of the goals are covered. If a test only covers one goal, then the name of the goal is an obvious candidate name for the test. Typically a test will

Table 5: Coverage goals for the test suite generated by EvoSuite (Figure 2); unique coverage goals are highlighted.

Test	Coverage Goals
`test0`	Method `createsShoppingCart` Method `addPrice` Output `addPriceReturningFalse` Input `addPriceWithPositive`
`test1`	Method `createsShoppingCart` Method `addPrice` Exception `addPriceThrowsIllegalArgumentException` Input `addPriceWithZero`
`test2`	Method `createsShoppingCart` Method `addPrice` Output `addPriceReturningTrue` Input `addPriceWithPositive`
`test3`	Method `createsShoppingCart` Method `getTotal` Output `getTotalReturningZero`

```
@Test
public void test0 / testAddPriceReturningFalse() {
   ShoppingCart cart0 = new ShoppingCart();
   boolean boolean0 = cart0.addPrice(2298);
   assertEquals(0, cart0.getTotal());
   assertFalse(boolean0);
}

@Test
public void test1 / testAddPriceThrowsIllegalArgumentException() {
   ShoppingCart cart0 = new ShoppingCart();
   // Undeclared exception!
   try {
    cart0.addPrice(0);
    fail("Expecting exception: IllegalArgumentException");

   } catch(IllegalArgumentException e) {
     // Cost cannot be negative
     verifyException("ShoppingCart", e);
   }
}

@Test
public void test2 / testAddPriceReturningTrue() {
   ShoppingCart cart0 = new ShoppingCart();
   boolean boolean0 = cart0.addPrice(1);
   assertEquals(1, cart0.getTotal());
   assertTrue(boolean0);
}

@Test
public void test3 / testGetTotal() {
   ShoppingCart cart0 = new ShoppingCart();
   int int0 = cart0.getTotal();
   assertEquals(0, int0);
}
```

Figure 2: Test suite generated with EvoSuite for class `ShoppingCart`, with original names and descriptive names.

cover multiple coverage goals at the same time, and the ideal name will be a combination of these goals. However, naively joining up goal names would easily result in too long names, and goals covered by multiple tests would make it more difficult to distinguish tests (R2). Therefore, this section describes how to create shorter, representative names given a test and the coverage goals it covers.

Algorithm 1 Test Name Generation

Input: Test suite T
Input: Set of coverage goals G
Input: Maximum number of goals in a name n

```
1:  procedure NameTests(T, G)
2:    for all t ∈ T do
3:      U ← CoveredGoals({t}, G) \ CoveredGoals(T \ {t}, G)
4:      top ← TopLevelGoals(U, n)
5:      name ← MergeText(top)
6:      Label(t, name)
7:    end for
8:    for all T' ⊂ T where all t ∈ T' have the same name do
9:      ResolveAmbiguities(T')
10:   end for
11:   for all t ∈ T do
12:     if t has no name then
13:       C ← CoveredGoals({t}, G)
14:       Label(t, Head(TopLevelGoals(C, n))
15:     end if
16:   end for
17:   for all t ∈ T do
18:     SummarizeText({t})
19:   end for
20:   for all T' ⊂ T where all t ∈ T' have the same name do
21:     AddSuffixes(T')
22:   end for
23: end procedure
```

As a running example for the naming process, Figure 2 shows a test suite generated automatically using EvoSuite [13]. Table 5 lists the coverage goals for each of these tests. Note that `test0` and `test2` call `getTotal` but these are not listed as method coverage goals for these tests. We ignore these calls since they are not part of the sequence of calls a test generator would produce, but within assertions which are usually added in a post-processing step.

3.3.1 Identification of unique goals. In a scenario where tests are generated with the intent of covering code, it is reasonable to assume that each test in a test suite covers something that no other test in the test suite covers; a test that does not provide any additional coverage would not be retained during test generation, or else would be removed during post-processing of generated test suites. Under this assumption, each test will have at least one *unique* coverage goal that distinguishes it from all other tests, which will influence the name given to the test. Note that even without a unique coverage goal our approach will still come up with a descriptive name; however, if a test cannot be distinguished in terms of coverage, then whether or not the test can be clearly distinguished from other tests in the same test suite by name will be a result of chance.

Given a set of covered goals for each test in a test suite, determining the set of unique covered goals is a matter of calculating the set complement of goals covered by a test, and goals covered by all other tests (set U in Algorithm 1). For example, considering the coverage goals listed in Table 5, the test `test0` is the only test to cover the output goal `addPriceReturningFalse`; `test1` is the only test to cover

exception goal `addPriceThrowsIllegalArgumentException` and input goal `addPriceWithZero`; `test2` uniquely covers the output goal `addPriceReturningTrue`; and `test3` uniquely covers the method `getTotal` and the output goal `getTotalReturningZero`.

3.3.2 Ranking test goals. A test will typically cover more than one goal. For example, if a method is covered, then it's return value will result in an output coverage goal that is covered, and any parameters will result in covered input coverage goals. To make sure that the name reflects only the minimum amount of information necessary to satisfy the three desired properties of a good name, we sort coverage goals based on the following ranking:

(1) Exception coverage is at the highest level in the hierarchy. Intuitively, if a test leads to an exception, then this is information we will want to see reflected in the name.

(2) Method coverage is ranked next; even if there is no exception, we always want a test name to describe at least the method it targets.

(3) Output coverage follows method coverage, since the output (return value) often represents the result of the scenario under test. Intuitively, if a test uniquely covers a method, then the method coverage is sufficient information. However, if there are several tests covering the same method, then the differences in how the method is called are captured by different return values.

(4) Input coverage represents information about the "scenario" of the test, and we consider this only if no other coverage type can be used to name a test.

Given the set of unique coverage goals for a test, the next step in the name generation is thus to select all goals of the highest rank (TopLevelGoals in Algorithm 1). That is, if there are exception goals covered, then these will be used for the name generation (for example, `test1` in Table 5). If not, then all method goals will form the basis for name generation (e.g., method `getTotal` for `test3`). If there are no uniquely covered methods, we next look at uniquely covered output goals, since the values produced are representative of the expected behavior (e.g., `addPriceReturningFalse` for `test0` and `addPriceReturningTrue` for `test2`). Finally, if there are no output goals, then we select all unique input goals.

Note that it is possible for a test to not uniquely cover any coverage goals; this might be because the test suite is not minimized and the test is redundant, or it might be that there are several tests that cover different lines, branches, or other coverage entities not contained in our hierarchy.

3.3.3 Merging test goals. Next, we convert the set of highest level unique coverage goals to text (MergeText in Algorithm 1). If there are no such coverage goals, then the initial name is an empty string. If there is exactly one goal, then the text representation of that goal is the name. For example, in Table 5, each test has exactly one top-level goal. If there is more than one goal, then we concatenate their text representation using the `And` keyword. During this concatenation, for any subset of coverage goals that target the same method (e.g., if a test covers two input coverage goals for the same method call), the method name will only be listed for the first one. If a coverage goal on a constructor is merged with a coverage goal on a method, then instead of just `And` these are merged using `AndCalls`; for example `testCreatesFooAndCallsBar`.

In order to avoid long names, we use a maximum of n goals for concatenation (in our case, $n = 2$). If there are more than n goals, we select a subset of n goals. For this selection we prioritize the coverage goals based on the methods they relate to. In particular, we prioritize methods that change the state of an instance of the class under test (i.e., impure method), and methods whose return values are involved in assertions. If there are more than n goals for the same method, we make a random selection of n goals.

3.3.4 Resolving ambiguities. At this point, each test has a name, but the name might not be unique, for example if two or more tests only differ in terms of lower level coverage (e.g., branches) and thus an empty name. For each ambiguous name we select all tests that resulted in that name (T'). For each of these tests, ResolveAmbiguities computes the set of unique coverage goals relative to the set of ambiguous tests (rather than the whole test suite), and then selects the top ranked goal out of that set.

If, at the end of this process, there exist tests that have no name, then we consider the set of non-uniquely covered goals for that test (C), and select the top ranked goal (Head(TopLevelGoals(C))). Now that all tests have names, we use a basic postprocessing technique based on abstractive text summarisation algorithms (SummarizeText). This step simplifies common patterns to more natural versions. For example, the following test name is based on a `List` returned by method `foo`:

`testFooReturningListWhereIsEmptyIsFalse`

For collection classes, we reduce the use of observers such that the resulting test is named as follows:

`testFooReturningNonEmptyList`

Similarly, we reduce output goals on Boolean values. For example, `testEqualsReturningFalse` is changed to the simpler version `testNotEquals`.

Finally, we append indices to all names in any remaining set of ambiguous test names to resolve duplication (AddSuffixes). Each test in Figure 2 shows the name resulting from the overall process.

4 EVALUATION

To evaluate the effectiveness of the proposed technique to generate descriptive names *for automatically generated unit tests*, we conducted an empirical evaluation aimed at answering the following research questions:

- **RQ1** Do developers agree with synthesized test names?
- **RQ2** Can developers match unit tests with synthesized test names?
- **RQ3** Can developers use synthesized test names to identify relevant unit tests for a given piece of code?

4.1 Experimental setup

4.1.1 Subjects. We recruited participants by direct email invitations to Computer Science and Software Engineering students at the University of — and the University of —. All participants were prescreened using a Java and JUnit qualification quiz: they were required to answer correctly at least 3 out of 5 competency questions. As a result, a total of 47 unique applicants were accepted and took part in the study.

4.1.2 Treatments. We consider two treatments: test names synthesized by our coverage-based naming technique (*synthesized*) and names manually written by experienced developers (*manual*).

4.1.3 Tasks. The three research questions in the study are addressed by assigning participants the following tasks:

Agreement (RQ1). Given the code of a JUnit test and a suggested test name, indicate your level of agreement with the name.

Selection (RQ2). Given the code of a JUnit test, select the more appropriate, descriptive name from a given list of candidate names.

Understanding (RQ3). Given the source code of a Java class and a list of candidate test names, select the test name you think will execute a given line number in the source code.

4.1.4 Objects. We followed a systematic protocol to select objects from the SF110 corpus of open source Java projects [15]. SF110 consists of a statistically representative sample of 110 projects and includes 23,886 classes with more than 6.6 millions of lines of code in total. Our selection protocol consisted of the following steps:

- Download the SF110 compilable sources and tests package.
- Select classes for which a generated test suite exists in the package (i.e., for which test generation succeeds).
- Select classes with a ratio of at least two generated unit tests per public method. Test naming is particularly relevant when there is more than one test per public method.
- Generate test suites with descriptive, synthesized names for the selected classes.
- Select target methods for which there is strictly three generated tests. This is a necessary design decision. When there is only one (or two) tests for a target method, test naming is trivial (or nearly trivial). On the other hand, more than three tests per method would overly complicate the tasks (more and longer test names to analyze by participants) without adding value to the study.
- Run mutation analysis on the selected classes and test suites [23]. Select target methods for which at least one generated test uniquely kills at least one mutant. This is needed for the understanding task: a uniquely killed mutant provides the connection between a synthesized test name and a source code line.
- Finally, randomly select one target method per class, and for each target method, randomly select a test to be used to formulate agreement, selection and understanding tasks.

As a result of this systematic search, we selected ten target methods for ten different classes to be part of our study, each one with three automatically generated tests with descriptive, synthesized names. Table 6 summarizes this final object selection.

Baseline. We used test names manually created by experienced Java developers as baseline. Deciding this was the appropriate baseline to compare against was a challenging task. In principle, when looking at automatically generated tests, the obvious baseline should be enumerated test names – that is how generated tests are currently named after all. However, such comparison would be pointless at the very least: there would be no true value in improving over `test0` or `test1`. As an alternative, having seasoned human developers manually assign names to generated tests provides a realistic baseline and enables us to evaluate how the names synthesized using our coverage-based naming technique are perceived by developers other developers in a less contrived scenario. Under this rationale, we recruited two PhD students with extensive programming experience and knowledge of Java and JUnit and asked them to create this baseline dataset. We presented them with the 30 selected tests (three for each of the ten selected target method) and their respective classes under test. They were instructed to provide the most appropriate and descriptive name they could think of for all these tests. They worked independently and without time constraints. The resulting dataset was consolidated by two of the authors of this paper, who inspected each test and selected the best of the two names created by the experts. These names are shown in column "Manual names" of Table 6.

4.1.5 Procedure. The study was designed and implemented as an online survey. We designed webpage containing strictly the necessary information to complete each of the three tasks defined. The webpage for *agreement* tasks contains a tab panel for the target test, a tab panel for the target test shown in context with the other two tests for the same target method (the name of these context tests are concealed), an Likert-like input component (strong disagree, disagree, neutral, agree and strong agree) (Figure 3). The webpage for the *selection* tasks is similar to the one for agreement tasks, except that the name of the target test is concealed, and three shuffled candidate names are presented as choices (plus "None"). The webpage for *understanding* tasks, in turn, shows the source code of a class under test instead of the test code, with a target line number highlighted (the line where a mutant is uniquely killed by a test). Similarly to the webpage for selection tasks, three candidate names plus "None" are presented as options. In the three webpages, a required free-text explanation field is included to elicit the reasoning behind each participant's answer.

Questions. With ten target methods (with a selected test, as highlighted in Table 6), two treatments (synthesized and manual), and three tasks (agreement, selection, understanding), a total of 60 questions were formulated for the study and loaded on the survey.

Assignment. Upon login on the website, each participant was assigned ten questions, one per test, using a balanced random sampling algorithm. That is, for each test, a participant can see either manual or generated test names, and one of the three types of questions. The order in which question are presented is randomized.

4.1.6 Analysis. All data collected in the study is processed using well tested R scripts. For agreement tasks, we present Likert scales to reason about levels of agreement as a whole and per individual test. For the selection and understanding tasks, bar plots are used to present correctness results (categories "Correct", "Don't know", and "Incorrect"). For the three tasks, we present boxplots and report on effect sizes and $p-$values for the comparison of response times for synthesized and manually written test names.

4.1.7 Threats to Validity.

Objects. The tests used in this study may not be representative of larger, more complex test suites. We alleviated this by using tests generated systematically selected from a statistically significant sample of open source Java projects.

Participants. Most participants were students and hence may not be representatives of real-world experienced programmers [9, 20, 24]. This is a common issue encountered when conducting human studies, since it is nearly impossible to reach the number of professional developers needed to perform any statistical test on

Table 6: List of experimental objects. NUKM stands for Number of uniquely killed mutants. For each class, the test selected for the study is highlighted. The other tests names are used to be presented as options for selection and understanding tasks.

Test ID	Project	Class.Method	Synthesized names	Manual names	NUKM
1	jsecurity	ValueListImpl.asBoolean	testAsBooleanThrowsNullPointerException	testNullList	0
			testAsBooleanReturningEmptyList	testListEmptyOnCreation	0
			testAsBooleanReturningNonEmptyList	**testAddNullValue**	**1**
2	vuze	PeerItem.convertSourceID	**testConvertSourceIDReturningZero**	**testConvertSourceIDTracker**	**1**
			testConvertSourceIDThrowsNullPointerException	testNullArgException	0
			testConvertSourceIDReturningNegative	testUnrecognisedSourceID	6
3	jsecurity	SimpleAuthorizingAccount.getSimpleRole	testGetSimpleRolesReturningEmptySet	testEmptySimpleRoles	0
			testGetSimpleRolesReturningNull	testNullSimpleRoles	0
			testGetSimpleRolesReturningNonEmptySet	**testMergeRoles**	**10**
4	squirrel-sql	StringUtilities.getByteArray	testGetByteArrayWithNull	testNullParameter	3
			testGetByteArrayWithEmptyArray	testParameterOfLengthZero	0
			testGetByteArrayReturningNonEmptyArray	**testNonZeroLengthNotNull**	**2**
5	glengineer	VWordPosition.equals	**testEquals**	**testWhetherAVWordPositionsIsEqualToItSelf**	**5**
			testNotEquals	testWhetherTwoDifferentVWordPositionsAreNotEqual	4
			testEqualsWithNull	testWhetherAValidVWordPositionIsEqualToNull	1
6	vuze	BitFlags.and	testAndThrowsNullPointerException	testAndNullPointerException	3
			testAndThrowsArrayIndexOutOfBoundsException	**testAndArrayIndexOutOfBoundsException**	**5**
			testAndReturningBitFlagsWhereSizeIsZero	testAndSameObjectEquals	14
7	noen	DaikonConstraint.stringFrom	**testStringFromReturningNonEmptyString**	**testConversionOfANonEmptyListIntoString**	**3**
			testStringFromReturningEmptyString	testConversionOfAnEmptyListIntoString	0
			testStringFromThrowsNullPointerException	testConversionOfANullListIntoString	0
8	jsecurity	StringUtils.hasText	testHasTextWithNull	testHasTextNullArg	4
			testHasText	testHasTextNonEmptyArg	4
			testNotHasText	**testHasTextEmptyArg**	**2**
9	vuze	StringPattern.digitWildcard	testDigitWildcardTakingNoArgumentsReturningNull	testSingleArgConstructor	8
			testDigitWildcardTakingNoArgumentsReturningNonNull	testTwoArgConstructorWildcard	11
			testDigitWildcardTakingCharacter	**testNonMatchingPattern**	**94**
10	jiprof	ClassWriter.visitAnnotation	**testVisitAnnotationWithNonEmptyStringAndFalse**	**testVisitNonVisibleAnnotation**	**4**
			testVisitAnnotationWithEmptyString	testVisitVisibleAnnotation	4
			testVisitAnnotationThrowsNullPointerException	testVisitAnnotationNullPointerException	1

their data. However, the most recent research on this issue threat can be mitigated by carefully scoping of experimental goals [38].

Participants in the study may tend to answer randomly, which imposes a threat on the validity of conclusions drawn from their data. We alleviated this threat by asking them to provide open-ended reasons for all their answers.

Context. We purposefully limited the context available to participants when answering the study questions. It may be the case that having other artifacts available would affect some participants's answers. However that would have come at the unknown cost of introducing confounding variables in the experimental set up.

4.2 Results

4.2.1 RQ1: Agreement with synthesized test names. Figure 4a summarizes the results for the agreement questions, and shows that there is generally more agreement than disagreement with synthesized test names. Over all answers for questions with synthesized names, 53% indicated some level of agreement, while only 27% showed some level of disagreement (top of Figure 4a). This suggests that participants perceived synthesized names as appropriate and descriptive for their respective unit tests.

Positive examples. The highest levels of agreement are observed for Tests 06 (75%) and for Test 02 (67%). In the case of Test 06, the synthesized name makes it clear that the test captures an exceptional behavior (`testAndThrowsArrayIndexOutOfBoundsException`), while for Test 02 the name reveals the expected output value of a call: `testConvertSourceIDReturningZero`. In both cases, the agreement answers were supported by positive free-text responses, e.g., *"The test name is perfect. It does exactly that, call the method and check if it returns zero"*, and relevant suggestions, e.g., *"The test name is a bit too long for me. I would have named it testArrayOutOfBound"*.

Negative examples. The highest levels of disagreement, on the other hand, was observed for Test 07 at 43%, and Tests 04 and 10, both at 38%. For the latter two tests, one of the reasons given by participants for disagreement is the length of the name. In the case of Test 07, although the name is nearly as long as the ones for Tests 04 and 10, no participant complained about the length, but rather considered the name confusing and insufficient. In spite of these cases, the general trend observed in the data is that participants only mildly and not very often disagree with synthesized names, with the best case being Test 02, with 0% level of disagreement.

Comparison to manually written names. The levels of agreement and disagreement observed for synthesized names constitute evidence of the effectiveness our coverage-based approach at generating descriptive test names, but to properly gauge the level of agreement we need to consider how the synthesized names compare to manually written test names. Figure 4a depicts this comparison for each of the tests used in our study, and aggregated for all. Overall, we observe similar levels of agreement, but less disagreement in the case of synthesized names (28% vs 39%).

Two of the synthesized names with the highest disagreement (the ones for Test 07 and Test 04) are also the worst-performing compared to manually given names. Both cases exemplify how developers are often capable of eliciting abstract knowledge and capturing it into their test names. The manually given name for Test 07, `testConversionOfANonEmptyListIntoString`, makes it explicit that the target method performs a type conversion from

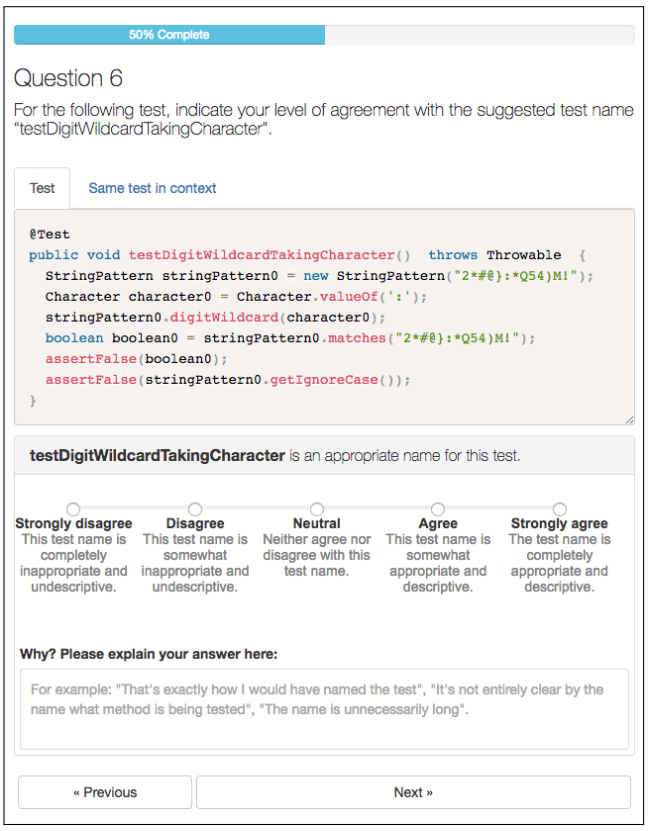
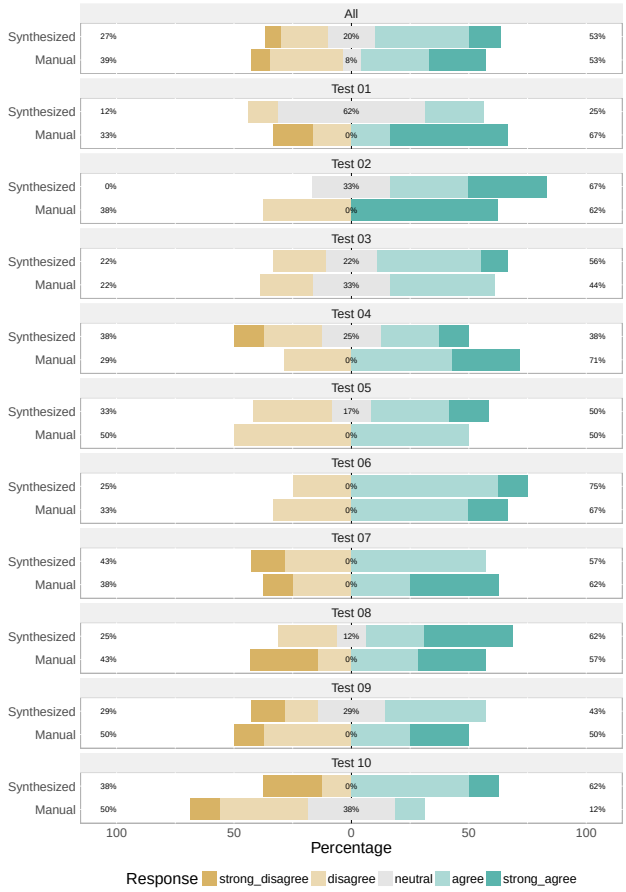


Figure 3: Example of agreement question as presented to participants in the survey website

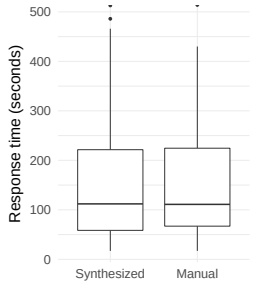
a populated list to a string object. It is not surprising that this name was more agreeable to participants than the synthesized `testStringFromReturningNonEmptyString`. Similarly, the manually given name `testNonZeroLengthNotNull` succinctly captures the flow of the test (not even mentioning the method being tested), and was more often agreed upon than its synthesized counterpart `testGetByteArrayReturningNonEmptyArray`.

We observe cases where participants clearly agreed more often with a synthesized name than with a manually given name, e.g., for Test 10 (manual `testVisitNonVisibleAnnotation` vs. synthesized `testVisitAnnotationWithNonEmptyStringAndFalse`). This case shows that the additional understanding developers can gather from the source code and build into their names is not always perceived as descriptive by other developers. Namely, the expert-given name `testVisitNonVisibleAnnotation` encodes information not evident in the test code itself (notion of non-visibility) which rendered the name *"unclear"*, *"unnecessarily long"* and *"poorly worded"*, according to participants.

For some cases, e.g., Test 02 and Test 08, participants showed agreement with both manual and synthesized names. For Test 02, for instance, although by different means, both test names are descriptive of what the test does. The manual name `testConvertSourceIDTracker`, captures a string constant value being used explicitly in the test code, whereas the synthesized name `testConvertSourceIDReturningZero` neatly describes the method under test and its expected return value.



(a) Likert agreement scales, aggregated at the top (labeled "All"), and individually for each of the ten tests in the study

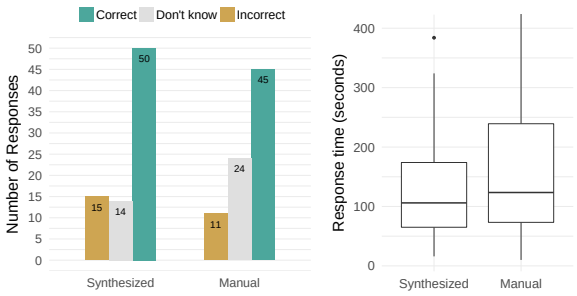


(b) Response times

Figure 4: Agreement Results (RQ1)

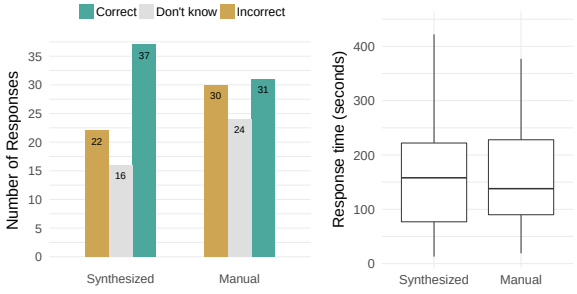
Timing. Besides agreement levels, the time it took participants to answer agreement questions is also indicative of how descriptive, hence how easy to interpret, test names are. Our timing results, Figure 4b, suggest that synthesized names are slightly easier to interpret than manually given names on average. The mean response time for synthesized names is 185.56″ compared to 222.87″ for manually given names, although there is no statistical significance between the two samples, with $A_{12} = 0.48$ and p–value=0.48.

***RQ1**: Participants in our study agreed similarly, and disagreed less, with synthesized test names than with manually given names.*



(a) Correctness **(b) Response time**

Figure 5: Selection Results (RQ2)



(a) Correctness **(b) Response time**

Figure 6: Understanding Results (RQ3)

4.2.2 RQ2: Matching unit tests with synthesized names. One of the key aspects of our approach is that it captures the uniqueness of each generated test in the context of a test suite to synthesize a descriptive name for it (cf. Section 3, R2). The selection tasks in our study aim to validate this focus on uniqueness by giving participants a choice of names and asking them to select the most descriptive for a given test. Figure 5 shows the results of these selection tasks in terms of correctness (Figure 5a) and response time Figure 5b. Our results show that participants were slightly more accurate at choosing the right name for a unit test when the name was synthesized, as opposed to manually created. A total of 50 correct selection answers were collected for synthesized names compared to 45 for manual ones. Furthermore, synthesized names seem to be indeed clearer in terms of uniqueness, as suggested by the lower number of *I don't know* answers (14 vs 24), although this positive observation is somewhat shadowed by the higher number of incorrect answers (15 vs. 11).

There is a also a trend indicating that participants were faster at selecting a synthesized test names. Although the difference is not statistically significant, it at least suggests that interpreting synthesized names does not represent extra effort to developers.

> ***RQ2**: Participants of our experiment were slightly more accurate and faster at matching tests and synthesized names.*

4.2.3 RQ3: Identifying relevant unit tests by names. Beyond descriptiveness and uniqueness, test names should expose the relation between the unit test and the code under test (cf. R3 in Section 3). In other words, a test's name should ideally help developers identify which parts of the code under test will be exercised by a test. This is exactly the scenario we present to participants in our *understanding* tasks. Figure 6 shows correctness and efficiency results for these tasks.

The positive trend observed for the selection tasks in favour of synthesized names is even more pronounced in the results for understanding tasks. As shown in Figure 6a, participants were more accurate at identifying the relevant tests for specific parts of the code when they were presented synthesized names. Synthesized names also reduced by a notable margin the number of incorrect and uncertain answers.

Being more accurate, in the case of understanding tasks, comes at a price of a marginally larger effort, as indicated by our timing results (Figure 6b). However, a statistical comparison indicates practically no difference between the two tasks, with (A_{12} = 0.49 and p-value=0.9.

> ***RQ3**: Participants of our study were more accurate at identifying relevant tests for given pieces of code using synthesized test names.*

5 CONCLUSIONS

There are many advantages to having well-named unit tests: Finding relevant unit tests becomes easier, the importance of test failures can be judged without reading the source code, and the test is easier to understand as the name summarizes its purpose. Automatically generated unit tests, however, often have no clear purpose, and thus generally are not given descriptive names.

In order to provide the benefits of well-named tests also to automated unit test generation, in this paper we have presented and evaluated an approach to derive names. Our extensive experiments showed that developers agree with the tests, and the names are equally descriptive as manually derived test names. We even noticed an improvement in the ability of study participants to match synthesized names with relevant tests and code. Consequently, our technique offers an easy and automated way to boost the usefulness of automatically generated tests.

The complete implementation of our coverage-based test naming technique has been developed as an extension to the open source EvoSuite test generation tool [13].

There is scope to improve the synthesized names further: In particular, our naming technique currently only uses assertions to rank coverage goals, but it would be possible to utilize them to derive better explanations of test purposes. A further possible improvement would be to adapt the synthesized names to the patterns and conventions used in the software project under test (cf. [19]). Furthermore, we plan to investigate whether our technique is also applicable for renaming manually written unit tests.

Method names are not the only identifiers involved in unit test generation; as future work we will investigate how generating useful variable names can influence test understanding and maintainability. It is also conceivable to extend our technique to synthesize not only short method names, but more elaborate explanations or summaries [34, 46].

REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 352–361, 2013.

[2] Agitar One, 2016. Last visited on 30.08.2016.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[4] N. Alshahwan and M. Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1345–1348. IEEE Press, 2012.

[5] J. H. Andrews, F. C. H. Li, and T. Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 144–153, New York, NY, USA, 2007. ACM.

[6] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: An evolutionary test approach for java. In *ICST*, pages 185–194. IEEE Computer Society, 2010.

[7] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 73–87, New York, NY, USA, 2000. ACM.

[8] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 169–180, New York, NY, USA, 2006. ACM.

[9] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *IEEE Int. Software Metrics Symposium*, pages 239–249, 2003.

[10] Codepro analytix, 2017. Last visited on 30.01.2017.

[11] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, Sept. 2004.

[12] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 107–118. ACM, 2015.

[13] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[14] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, 39(2):276–291, 2013.

[15] G. Fraser and A. Arcuri. A large scale evaluation of automated unit test generation using evosuite. 24(2):8, 2014.

[16] G. Fraser and A. Zeller. Exploiting common object usage in test case generation. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 80–89, 2011.

[17] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. on Software Engineering (TSE)*, 38(2):278–292, 2012.

[18] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 360–369. IEEE, 2013.

[19] E. W. Høst and B. M. Østvold. Debugging method names. In *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 294–317. Springer, 2009.

[20] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects—A comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.

[21] K. Inkumsah and T. Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 425–428. ACM, 2007.

[22] Parasoft JTest, 2016. Last visited on 30.08.2016.

[23] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436, 2014.

[24] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering (TSE)*, 28(8):721–734, Aug. 2002.

[25] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ISSRE '05, pages 267–276, Washington, DC, USA, 2005. IEEE Computer Society.

[26] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 417–420, New York, NY, USA, 2007. ACM.

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

[28] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. Grt: An automated test generator using orchestrated program analysis. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 842–847, Nov 2015.

[29] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. Grt: An automated test generator using orchestrated program analysis. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 842–847. IEEE, 2015.

[30] M. Oriol and S. Tassis. Testing .net code with yeti. In R. Calinescu, R. F. Paige, and M. Z. Kwiatkowska, editors, *ICECCS*, pages 264–265. IEEE Computer Society, 2010.

[31] R. Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[32] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[33] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. Automatic test case generation: what if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 130–141. ACM, 2016.

[34] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 547–558, New York, NY, USA, 2016. ACM.

[35] I. S. W. B. Prasetya. T3, a combinator-based random testing tool for java: Benchmarking. In T. E. J. Vos, K. Lakhotia, and S. Bauersfeld, editors, *FITTEST@ICTSS*, volume 8432 of *Lecture Notes in Computer Science*, pages 101–110. Springer, 2013.

[36] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *SSBSE 2015*, volume 9275 of *LNCS*, pages 93–108. Springer, 2015. Best Paper Award (Industry-relevant SBSE results).

[37] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, 41(3):294–313, 2015.

[38] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, volume 1, pages 666–676, 2015.

[39] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. *ACM SIGPLAN Notices*, 46(10):189–206, 2011.

[40] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. In *International conference on tests and proofs*, pages 134–153. Springer, 2008.

[41] P. Tonella. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29(4):119–128, July 2004.

[42] A. Trenk. Testing on the toilet: Writing descriptive test names. *Computer*, Oct. 2014.

[43] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*, pages 380–403. Springer, 2006.

[44] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM.

[45] B. Zhang, E. Hill, and J. Clause. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 625–636, New York, NY, USA, 2016. ACM.

[46] S. Zhang, C. Zhang, and M. Ernst. Automated documentation inference to explain failed tests. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 63–72, 2011.