Naming Amplified Tests Based on Improved Coverage

Nienke Nijkamp
Delft University of Technology
n.nijkamp@student.tudelft.nl

Carolin Brandt
Delft University of Technology
c.e.brandt@tudelft.nl

Andy Zaidman
Delft University of Technology
a.e.zaidman@tudelft.nl

***Abstract*—Test amplification generates new test cases that improve the coverage of an existing test suite. To convince developers to integrate these new test cases into their test suite, it is crucial to convey the behavior and the improvement in coverage that the amplified test case provides. In this paper, we present NATIC, an approach to generate names for amplified test cases based on the methods they additionally cover, compared to the existing test suite. In a survey among 16 participants with a background in Computer Science, we show that the test names generated by NATIC are valued similarly to names written by experts. According to the participants, the names generated by NATIC outperform expert-written names with respect to informing about coverage improvement, but lack in conveying a test's behavior. Finally, we discuss how a restriction to two mentioned methods per name would improve the understandability of the test names generated by NATIC.**

I. Introduction

Tools for automated test generation have been available for a few years, however they are still cumbersome to use for software developers [1]. One of the leading causes for the lack of their use is the readability of the generated test cases [2]. When a test is considered readable, it will be easier to perform tasks that require understanding it [3], such as locating the reason why the test fails [2] or using it as documentation [4]. The name of a test can be one of the most useful sources of information to understand a test [5], and therefore improving the name considerably improves a test's readability. Our goal for this paper is to investigate how one could automatically generate test names that convey the behavior and coverage contribution of amplified test cases.

Test amplification is the automatic process in which based on developer-written test cases, additional unit tests are generated that increase the instruction coverage of the existing test suite. We call these *amplified test cases* [6]. In this paper, we design and implement an approach for naming amplified test cases generated by the test tool DSpot [7], and its IntelliJ plugin TestCube[1] [8].

The test cases generated by TestCube and DSpot have generic and unclear names, which affects the readability of the test cases. In this paper, we examine existing approaches for automatically naming unit tests [5], [9], [10]. Based on the approach by Daka et al. [9] we design NATIC, an approach to *Name Amplified Tests based on Improved Coverage*, using the improvement in coverage to name the amplified test case. Daka et al. infer names for generated test cases from the exceptions, methods, outputs and inputs which are covered by the test cases. In contrast to this, NATIC is specialized for amplified test cases and uses the names of the methods where additional instructions are covered to construct the name for the amplified test case. Our aim is to clearly convey to the developer how the amplified test case impacts the coverage of the test suite.

We implement NATIC as an extension to TestCube, and conduct a survey to determine:

> **Research Question:** How readable are the test names generated by NATIC compared to names generated by DSpot or names given by experts?

In our survey, 16 participants with a background in Computer Science rated the names generated by NATIC, the original DSpot names and names written by experienced Java developers in terms of their appropriateness, how well they convey the behavior of the test case, as well as how well they convey the coverage improvement that the test provides. Our results show that the names generated by NATIC clearly outperform the existing DSpot implementation. They slightly outperform the expert-given names in terms of conveying the coverage improvement, but are slightly worse in conveying the behavior of the studied test cases. Reflecting in more detail on the studied test names, we conclude that focussing on names based on at maximum two additionally covered methods would yield more readable test names.

In summary, this paper makes the following contributions:

- NATIC, an approach to generate understandable names for amplified tests, based on their coverage improvement.
- A study evaluating the approach against the names from DSpot, and against names written by experts.
- A replication package[2] that includes the implementation of NATIC in TestCube and the data from our evaluation.

II. Background

Writing unit tests for their software is one of the central tasks for a developer to deliver high-quality software [11]. Unit tests check the correctness of units of a program in isolation [12]. However, to this day, software developers consider

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032)

[1]https://github.com/TestShiftProject/test-cube

[2]https://doi.org/10.6084/m9.figshare.14909850

writing them a laborious, tedious and often difficult task [13]. This can be even more difficult when the entire testing program has to change when the code under test is changing. For this reason, several automated unit test generation tools exist, such as EvoSuite [14] and TestFul [15]. For this paper, we provide an improvement to the test amplification tool TestCube.

A. Amplifying Tests with TestCube

Software developers that want to improve their test suite, can use automated test amplification. TestCube is an IntelliJ plugin that uses the test amplification process of DSpot, which improves the original test by triggering new behaviors and adding new assertions [7]. The result of amplifying a test case with DSpot is a set of new test cases that cover previously not tested instructions. The tests generated by DSpot still have usability issues, the lack of understandable test names being an important one.

```
1  // Original Test Case
2  @Test public void testId() {
3      Document doc = Jsoup.parse("<div id=Foo>");
4      Element el = doc.selectFirst("div");
5      assertEquals("Foo", el.id()); }
6  // Amplified Test Case
7  @Test public void testId_assSep8() throws Exception {
8      Document doc = Jsoup.parse("<div id=Foo>");
9      Element el = doc.selectFirst("div");
10     assertFalse(doc.hasText()); }
```

Listing 1: Original Test Case and Amplified Test Case from DSpot

Listing 1 shows an original test case from example project jsoup[3], and an amplified test case generated by DSpot. Line 7 contains the current naming of DSpot.

B. Automatically Generating Names For Test Cases

In this paper, we define test names as *good* if they describe the intent of the test case, and increase the readability for the developer working with the amplified test cases. Benefits of descriptive names are:

- Ease of identifying which functionality the test checks.
- Documenting the class under test, the names of the tests can identify the supported functionality of the class.

Several approaches automatically generate names for unit tests, all using different features of the tests they are naming. This section compares 3 approaches: NAMEASSIST by Zhang et al. [5], DEEPTC-ENHANCER by Roy et al. [10], and the approach by Daka et al. [9]. From this comparison, we extract the desired features for the implementation for TestCube. The approach by Daka et al. does not have a name, we identify this approach as DAKANAMING for the remainder of this paper.

1) NameAssist by Zhang et al.: NAMEASSIST is an approach proposed by Zhang et al. [5], that creates descriptive test names using two phases: an analysis phase and a text generation phase. The approach rates three key aspects of each test: the action under test (usually the class under test), the expected outcome (the assertion under test) and the scenario under test (the body of the test). These three aspects are combined in the text generation phase.

2) DakaNaming by Daka et al.: Daka et al. propose an approach that extracts coverage goals from generated test cases and ranks them to generate a unique test name [9]. The approach relies on the insight that the context of the test suite provides information to derive descriptive names that link the source code to the test name. The coverage goals are ranked according to how observable their impact is for the developer: 1) Exception Coverage, 2) Method Coverage, 3) Output Coverage and 4) Input Coverage. The authors conducted a study with 47 participants, showing that the generated names are as descriptive as manually written ones.

3) DeepTC-Enhancer by Roy et al.: DEEPTC-ENHANCER is an approach proposed by Roy et al. [10], which generates descriptive identifiers for generated test cases and test method-level summaries of test case scenarios. This is achieved by using existing code summarization approaches and deep learning techniques. This approach addresses the problem of lack of documentation and the meaningless identifiers (test and variable names) for generated test cases.

Summary. NAMEASSIST relies on descriptive variable names, which are not present in the amplified tests, and DEEPTC-ENHANCER has a complex structure and functionalities not necessary for our research, such as generating documenting comments and variable names. Therefore, we use DAKANAMING as the blueprint for our approach and adapt it to fit test amplification. We use the methods where an amplified test covers additional instructions to name the test.

III. NATIC

NATIC is centered around the idea that the test case name of the amplified test should tell the developer what the test case contributes to the overall test suite. Considering that NATIC uses the amplified test cases from DSpot as input, and knowing that DSpot does not generate test cases with identical coverage improvement, we use the *coverage improvement* to generate new test cases names. Specifically, NATIC uses the coverage goals as identifiers for every test case, and generates a unique name for every amplified test case in the test suite.

A. Coverage Goals

TestCube reports the coverage improvement for every test it generates. NATIC extracts the methods from the test case report to initialize them as coverage goals for the remainder of the program (COVEREDGOALS in Algorithm 1).

B. Why Additional Method Coverage?

The name of a test case should describe and summarize important parts of the test's body [16]. In test amplification, the additional coverage symbolizes the additional value the test will bring to the existing test suite. To convey this additional value and point the developer to the methods which are only covered by this test—likely the fault location if this test fails—we use the additionally covered methods to name the test case.

[3]https://github.com/jhy/jsoup

Algorithm 1: NATIC

```
Input: Amplified Test Suite = T
forall t ∈ T do
    goals ← CoveredGoals({t}, G) \ CoveredGoals(T \ {t}, G)
    name ← MergeText(t, goals)
    Label(t, name)
forall t ∈ T do
    if t has no name then
        C ← CoveredGoals({t}, G)
        C ← RemoveDuplicateGoals({t}, G)
        Label(t, UniqueGoals(C))
forall T' ⊂ T where all t ∈ T' have the same name do
    FixAmbiguousNames(T')
```

C. Finding Unique Names

An amplified test can produce multiple coverage goals, as it can improve the coverage in multiple methods, and multiple amplified tests can have the same coverage goal. If there is exactly one coverage goal for the test, the test name will have this coverage goal. We find unique goals per test by taking the complement of the goals covered by a test, and goals covered in all generated tests (*goals* in Algorithm 1). If a test covers the same method multiple times, the duplicate method names are removed (RemoveDuplicateGoals in Algorithm1).

We label each test case by concatenating its goals (Label in Algorithm 1). For each non-unique name we take all tests that result in this name (T'). FixAmbiguousNames adds numerical suffixes to the test names if there are still duplicates after identifying unique goals and removing duplicate goals.

IV. Evaluation Setup

In this section we illustrate our study to evaluate NATIC. We compare the names generated by `DSpot`, `Expert` written names, and the names generated by NATIC.

A. Subjects

We employed convenience sampling for participant recruitment by publishing a call for participation on various platforms (Twitter, LinkedIn, etc.). After the participants agreed to participate in the study and acknowledge to their data being collected, we asked them whether they have a background in Computer Science. For this question, we specified a background in Computer Science as being able to understand source code and unit tests. Ultimately, 16 participants took part in the study.

B. Tasks

Given the original test case, the amplified test cases and their names, the participants were asked to rate their level of agreement for every amplified test case for these aspects:

- *Appropriate Name*: is an appropriate name for the test.
- *Behavior*: gives information on the behavior of the test.
- *Coverage Improvement*: gives information on the coverage improvement of the test.

C. Objects

We compare three kinds of test names: (1) names generated by the current implementation (`DSpot`), (2) names generated by our method-coverage based approach (NATIC), and (3) names written by experienced software developers/testers (`Expert`). We obtained the test names used in the study from running DSpot on three unit tests from the open-source HTML parser jsoup. The result from the amplification consists of the amplified test cases with their names, `DSpot` or NATIC. The `Expert` names were written by a Computer Science BSc student and a PhD student, both with extensive programming experience and knowledge of software testing. We showed the experts the original test case, and the amplified test case without a name. Table I gives examples of the names used in the study, and it also lists the test case's coverage improvement.

D. Procedure

The study was performed as an online survey. Every participant got the same sample of 25 test cases, with names from the three different sources. The order of the types of test names was randomized. The participants were given the tasks from Section IV-B, they could add feedback to either the survey or the test names at the end of the survey. The processed data leads to the distributions shown in Figures 1, 2, 3. For every test case, the participants added one answer, in a 5-point Likert-scale (Strongly disagree, Disagree, Neutral, Agree, Strongly agree).

V. Results

In this section, we illustrate the results of our survey and present the agreements for `DSpot`, `Expert`, and NATIC.

A. DSpot

Figure 1 represents the results for the DSpot names. Overall answers for these test names resulted in a high level of disagreement (either Disagree or Strongly disagree). For the appropriate name, the level of some disagreement was 91%, for information on the behavior of the test the level of disagreement was 80%, and for information on coverage improvement it was 91%. This indicates that participants considered the test names neither appropriate nor informative.

B. Expert

Figure 2 represents the results for the names manually written by experts. The agreement with these names is significantly higher than the DSpot names. The `Expert` names get 83% for some level of agreement (either Agree or Strongly agree) on whether the name is appropriate, 80% on information on the behavior of the test, and 79% on the information of coverage improvement of the test. These are high acceptance rates, the participants liked and understood these test names.

Coverage Improv. for Methods	DSpot	NATIC	Expert
hasText outputSettings	dropSlashFromAttribute Name-mg43-assSep103	testOutputSettingsAndHasText	testSetOutputSettingsWithText
clone doClone	filter-mg69-assSep208	testCloneAndDoClone	testCloneEmptyElement
documentType	testGetChildText-mg30-assSep223	testDocumentType	testDocumentTypeIsNull

TABLE I: Examples of test names used as objects for the survey

C. NATIC

Figure 3 represents the results for the names generated by NATIC. The results for NATIC are slightly more spread-out when compared to the other two types of names. Since the preference for the names from this approach was more spread, Figure 4 shows a breakdown per test.

For the NATIC test names, the level of (some) agreement regarding appropriate name was 54%, for the information on behavior 57%, and for the information on coverage improvement 81%. The participants agreed that the test names give information on the coverage improvement.

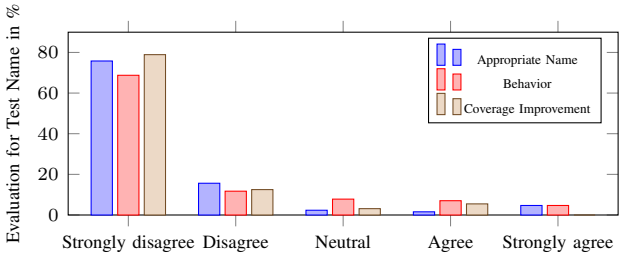

Fig. 1: Distribution for `DSpot`

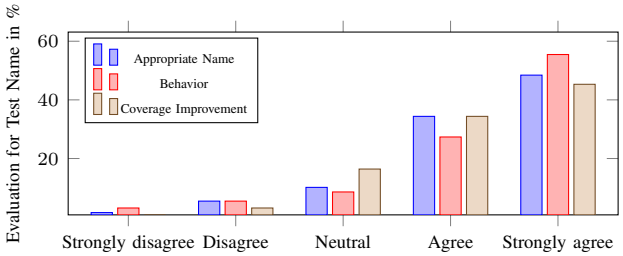

Fig. 2: Distribution for `Expert`

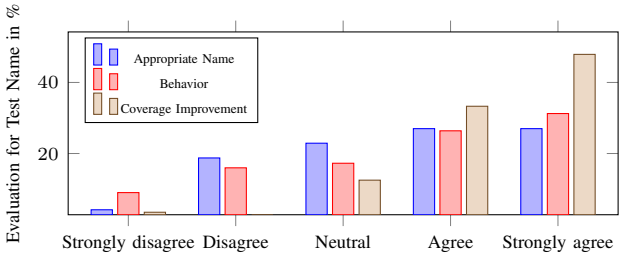

Fig. 3: Distribution for NATIC

D. Free-text Responses

All participants could give additional feedback on the names given to the tests. Most responses concerned the length of the survey, participants perceived the survey as repetitive and too long. Some participants had tips for naming the tests:

- *"I usually name my tests after the direct actions, the tests do. When I found myself disagreeing with names it was often because the name was related to the action that was taken by the test indirectly."*
- *"The test names which talked more about the function of the test were more helpful than either the ones that just listed the improvements in coverage or the ones that had a random name of a string."*

These participants needed more information than just the coverage improvement and suggested to add information, like the input to the test or the action taken by the test.

VI. Discussion

From the individual results per test in Figure 4, we can see that Test 5 and Test 9 (see Listing 2) have a particularly high disagreement rating. These test names have a common denominator, they were the only test names from NATIC that had more than 2 goals in their name. Test 3 (see Listing 2) had a particularly high agreement rate, which leads to our assumption that using more than 2 goals negatively impacts the agreement with a test name.

```
// Test 3, Improves Coverage in: toString, outerHtml
@Test public void testToStringAndOuterHtml() throws
      Exception {}

// Test 5, Improves Coverage in: rewindToMark, cacheString,
      consumeCharacterReference
@Test public void testRewindToMarkAndCacheStringAnd
ConsumeCharacterReference() throws Exception {}

// Test 9, Improves Coverage in: padding, isWhitespace,
      indent, siblingIndex, nodenames, outerHtmlHead, ...
@Test public void
      testPaddingAndIsWhitespaceAndIndentAndSiblingIndex()
      throws Exception {}
```

Listing 2: Test Names Generated by NATIC

The participants of the study collectively disagreed with the names given by `DSpot`. The `Expert` names had a higher agreement rating than NATIC. Based on our results and the distribution shown in Figure 4, we hypothesize that names generated by NATIC should contain at most 2 coverage goals. Future research has to confirm this hypothesis.

Answer to the Research Question: The results from the experiment indicate that the names generated by NATIC are effective at indicating the coverage improvement of an amplified test case and, with a limitation to two method names at most, are effective at generating appropriate names and give information on the amplified test case.

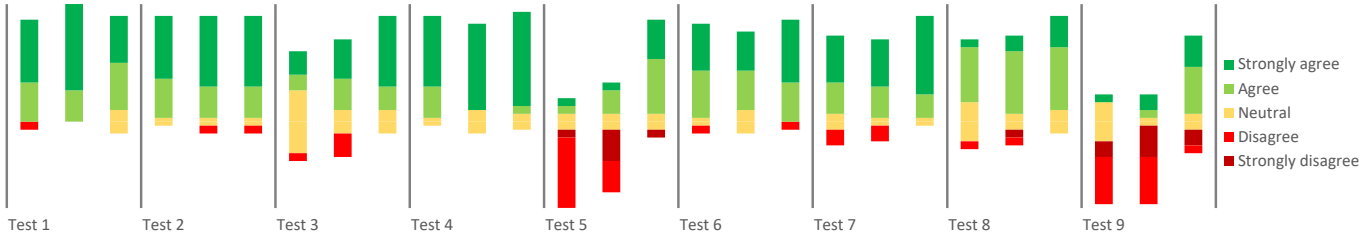

Fig. 4: Likert agreement for each of the tests from NATIC

A. Threats to Validity

1) Construct Validity: The survey was distributed through several platforms, and the only validation of the participants' expertise was their answer regarding it. We did not distinguish between students and professional software developers regarding their background in Computer Science. This risk is mitigated through existing research stating the lack of difference in results from Software Engineering students and professional software developers [17].

2) Internal Validity: Despite increasing the readability of the test cases by adding understandable names to them, the variable names were generated by DSpot. This could affect the readability of the overall test cases and therefore the understanding of the test case. To mitigate this threat would mean possibly tainting the results of the study, and was not beneficial to the study. Another threat could be 'cheaters', participants entering random data and with that skewing the results. This was mitigated by checking the response time for all participants. Cheaters take a significantly smaller time to answer questions in a survey [18], and the participants in our study all took at least the estimated 20 minutes.

3) External Validity: The methods and tests from jsoup might not reflect larger, more complex test cases. However, DSpot generates the same type of tests for every kind of test that is fed into it, so that the experiment can be replicated with larger test cases and still give the same output.

VII. Conclusions and Future Work

Amplified test cases benefit from understandable and descriptive test case names, as "good" names make it easier on developers to identify the functionality the test checks and documenting the class under test.

In this paper, we have presented the NATIC approach, which is based on the approach by Daka et al. [9], however NATIC names the tests according to the *additionally covered methods*. Our study shows that the generated names were a clear improvement over the default naming from DSpot. We have also identified additional constraints on the amount of goals in the names that can potentially further improve NATIC in the future. Furthermore, NATIC currently only generates test cases names for amplified test cases, we aim to further improve the readability of test cases by also adding descriptive variable names to the amplified test cases.

References

[1] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *Proc. Int'l Conf. on Software Engineering: Software Engineering in Practice Track*. IEEE, 2017, pp. 263–272.

[2] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, p. 107–118.

[3] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1100–1125, 2014.

[4] K. L. Beck, *Test-Driven Development - By Example*, ser. The Addison-Wesley signature series. Addison-Wesley, 2003.

[5] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 625–636.

[6] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[7] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus, "Automatic test improvement with DSpot: a study with ten mature open-source projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2603–2635, 2019.

[8] C. Brandt and A. Zaidman, "Developer-centric test amplification: The interplay between automatic generation and human exploration."

[9] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 57—67.

[10] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "DeepTC-Enhancer: Improving the readability of automatically generated tests," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2020, pp. 287—-298.

[11] J. Link, *Unit testing in Java: how tests drive the code*. Elsevier, 2003.

[12] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An empirical comparison of automated generation and classification techniques for object-oriented unit testing," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 2006, pp. 59–68.

[13] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The jml and junit way," in *European Conference on Object-Oriented Programming*. Springer, 2002, pp. 231–255.

[14] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proc. of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 416—419.

[15] L. Baresi and M. Miraz, "Testful: automatic unit-test generation for java classes," in *Proc. Int'l Conf on Software Engineering (ICSE)*, 2010, pp. 281–284.

[16] B. Zhang, E. Hill, and J. Clause, "Automatically generating test templates from test names," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 506–511.

[17] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2015, pp. 666–676.

[18] F. Rogers and M. Richarme, "The honesty of online survey respondents: Lessons learned and prescriptive remedies," *Decision Analyst, Inc White Papers*, pp. 1–5, 2009.