Developer-Friendly Test Cases: Generating Understandable Test Names Based on Coverage Improvement

Nienke Nijkamp[1] , **Carolin Brandt**[1] , **Andy Zaidman**[1]
[1]TU Delft

Abstract

TestCube amplifies existing unit tests and creates a new test suite with additional coverage for the source code. The names automatically generated by TestCube do not give any information on the behaviour or the coverage improvement of the amplified test case. In this paper, we present an approach to naming these amplified test cases by representing methods where the coverage is improved. These tests represent the covered methods on source code level and give the developer increased readability and understanding of the amplified test cases. We conducted a research study amongst 16 participants with a background in Computer Science. Participants were asked to indicate their agreement with the original test names, the test names generated by the approach and test names written by experts. The study found that participants strongly disagreed with the original TestCube names, and the names generated by our approach posed a real improvement to their satisfaction with the test names. With a few improvements, the test names generated by the approach will perform as acceptable as the manually written names.

1 Introduction

Tools for automated test generation have been available for a few years, however have not yet been widely used by software developers [1]. One of the leading causes for the lack of use is the readability of the generated test cases [1]. The readability of tests is the visual appearance of the code in general. When a test is considered readable, it will be easier to perform tasks that require understanding it [2]. The name of a test can be one of the most useful sources of information for understanding a test [3], and therefore improving the name considerably improves the test's readability. For this paper, we implement an approach for naming amplified test cases for TestCube.

TestCube[1] is an IntelliJ plugin that uses the test amplification process of DSpot [1]. DSpot is a tool that automatically generates unit tests from developer-written tests. We refer to the tests generated by TestCube and DSpot by the term amplified test cases. DSpot generates new assertions from the manually written test to create amplified test cases for previously untested scenarios.

The test cases generated by TestCube have generic and unclear names, which affects the readability of the test cases. We examine several existing approaches for naming unit tests to identify key features of every approach. We use these key features to implement a naming approach to TestCube. The first research question aims to summarize and compare existing approaches [3–5] for naming test cases by developer friendliness and readability.

> RQ1
>
> What are the existing approaches for naming generated test cases?

The names of the tests created by TestCube need to be improved to enhance the readability of the tests. The existing approaches have benefits and drawbacks, and we implement an approach in TestCube. We want to determine whether the prototype implementation improves the readability for developers. We call our prototype implementation `NATIC`, Naming Amplified Tests by using Improved Coverage.

> RQ2
>
> How readable are the test names generated by `NATIC` compared to the names generated by TestCube or names given by experts?

In this paper, we present an approach to naming automatically generated test cases. `NATIC` takes the coverage improvement of every individual amplified test case and creates unique test names for every test suite. We base this on an approach proposed by Daka et. al [4] taking different goals for creating the test names.

We conducted a user study amongst participants with a background in Computer Science. We evaluated the current test names, the test names generated by the new approach, and test names manually written by an expert. The manually written names are compared in the study for their readability, since most manually written names are significantly more

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

readable than generated ones [6].

This paper makes the following contributions:

- An approach to generate understandable names for amplified test cases, based on their coverage improvement.
- `NATIC`, a prototype implementation of the approach as an extension to TestCube.
- A study evaluating the approach against the original naming of TestCube, and against test names manually written by experts.
- A replication package[2] that includes (1) the implementation of `NATIC`, (2) the conducted research study, and (3) the data used for evaluation.

2 Background

Writing unit tests for their software is one of the central tasks for a developer to deliver high-quality software [7]. Unit tests check the correctness of units of a program in isolation [8]. However, to this day, software developers consider writing them a laborious, tedious and often difficult task [9]. 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 [10] and TestFul [11]. For this paper, we provide an improvement to the test amplification tool TestCube.

2.1 TestCube

Software developers that want to improve their test suite, can use automated unit test generation and amplification. TestCube is an IntelliJ plugin that uses the test amplification process of DSpot, which improves the original test by triggering new behaviours and adding new valuable assertions [12]. The result of amplifying an original test case with TestCube is a test suite with several new test cases. TestCube does still have a few usability issues, the lack of understandable test names being an important one [1].

```
1  // Original Test Case
2  @Test
3  public void testId() {
4      Document doc = Jsoup.parse("<div id=Foo>");
5      Element el = doc.selectFirst("div");
6      assertEquals("Foo", el.id());
7  }
8  // Amplified Test Case
9  @Test
10 public void testId_assSep8() throws Exception {
11     Document doc = Jsoup.parse("<div id=Foo>");
12     Element el = doc.selectFirst("div");
13     assertFalse(doc.hasText());
14 }
```

Listing 1: Original Test Case and TestCube Amplified Test Case

Listing 1 shows an original test case from example project jsoup [3], and an amplified test case generated by TestCube. Line 11 contains the current naming of TestCube.

[2] doi:10.5281/zenodo.5032953

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

2.2 Test Naming

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. [3], DeepTC-Enhancer by Roy et al. [5], and the approach by Daka et al. [4].

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 Daka 2017 for the remainder of this paper.

NameAssist by Zhang et al.

NameAssist is an approach proposed by Zhang et al. [3], 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, the expected outcome and the scenario under test. The action under test is usually the class under test. The expected outcome has the (single) assertion under test. The scenario under test is the body of the test. These three aspects are combined in the text-generation phase.

This approach is not applicable to TestCube for multiple reasons:

- The description of the test relies on descriptive variable names, which the amplified test cases do not have.
- The amplified test cases do not just check one assertion, they check several, therefore the expected outcome has too little information.
- The question which of the three aspects to use remains unanswered.

The implementation and documentation of this approach are not widely available.

Daka 2017 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 [4]. The approach relies on the insight that an individual generated test case might not have a clear purpose. The context of the test suite does provide sufficient information to derive descriptive names that link the source code to the test name. The coverage goals generated from the test suite are ranked according to how observable their impact is for the developer:

1. Exception Coverage
2. Method Coverage
3. Output Coverage
4. Input Coverage

Daka 2017 solely focuses on taking a test suite and generating unique names for the test cases, which is a feature desirable for the implementation for TestCube.

A study amongst 47 applicants was done by the authors, in which the applicants were asked to rate their agreement with generated test names compared to manually written test names. The applicants agreed similarly with both and disagreed with the generated names less than they did with the manually written names. These results show the effectiveness of this approach and suggest that the test names generated are considered descriptive of the generated test cases.

DeepTC-Enhancer by Roy et al.

DeepTC-Enhancer is an approach proposed by Roy et al. [5], which generates descriptive identifiers for generated test cases and test method-level summaries of test case scenarios. DeepTC-Enhancer automatically generates method-level summaries and renames identifiers. This is achieved by using existing code summarization approaches and deep learning techniques shown in Figure 1. This approach addresses the problem of lack of documentation and the meaningless identifiers (test and variable names) for generated test cases.

DeepTC-Enhancer changes the variable and test names and does not rely on the variable names in the source code. The approach is complex and is not solely aimed at improving test names. Therefore, it poses a larger challenge to adapt to TestCube, and it might be a valuable addition to TestCube in the future.

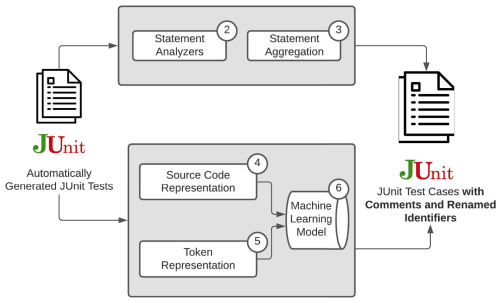

Figure 1: Overview of DeepTC-Enhancer - Reused from Roy et al. [5]

Answer to RQ1

NameAssist has characteristics that make it suboptimal for implementation in TestCube. DeepTC-Enhancer is effective at naming generated test cases. However, due to the complex structure of the program and the additional functionalities not necessary for this research, DeepTC-Enhancer will not be implemented to TestCube for this research.
Daka 2017 generates descriptive names according to developers. Considering that Daka 2017 uses a test suite as input makes it adaptable to the test suite generated by TestCube.
We will base the new approach on Daka 2017.

3 NATIC

We use Daka 2017 as the blueprint for our approach, which has an implementation to Evosuite [10]. We adapt this approach to fit TestCube. The methods in NATIC are based on the methods from the Evosuite implementation [4].

The new approach is based on coverage improvement of the amplified test cases. We call the proposed approach NATIC, Naming Amplified Tests by using Improved Coverage.

NATIC uses the test suite generated by TestCube from an original test case as input. TestCube only selects the test cases which improve *additional* instructions on any line [1], consequently the test suite will not contain tests with identical coverage improvement. The coverage goals are the names of the methods in which the coverage is improved. Using the coverage improvement instead of coverage tells the developer what the test case contributes to the overall test suite. NATIC uses these coverage goals as identifiers for every test case, and generates for every amplified test case in the test suite a unique name.

3.1 Coverage Goals

TestCube automatically generates a coverage improvement report for every test suite it generates, consisting of an extended coverage improvement for every amplified test case. Therefore, for every amplified test case, the methods that are additionally covered are embedded in this report. NATIC extracts the methods from the test case report to initialize them as coverage goals for the remainder of the program (labelled CoveredGoals in Algorithm 1).

3.2 Why Method Coverage?

The name of a test case should describe and summarize important parts of the test's body [13]. The methods a test calls are the most general description of the test's behaviour. Developers can quickly identify faults in the method calls if the name consists of the methods it improves.

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')
```

[4] https://github.com/EvoSuite/evosuite/blob/master/client/src/main/java/org/evosuite/junit/naming/methods/CoverageGoalTestNameGenerationStrategy.java

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

Table 1: Examples of test names used as objects for the survey

3.3 Finding Unique Names

An amplified test can contain multiple coverage goals, 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. Finding unique goals per test is a matter of taking the complement of the coverage goals covered by a test, and goals covered in all the tests (labelled *goals* in Algorithm 1). If a test covers the same method multiple times, the duplicate method names are removed (labelled REMOVEDUPLICATEGOALS in Algorithm1).

3.4 Resolving Ambiguities

Every test should have a name, however the name might not be unique. 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.

4 Experimental Setup

In this section we illustrate the research study we did to evaluate `NATIC`. We compare the test names generated by the `TestCube Implementation`, `Expert` written names, and the test names generated by `NATIC`. We asked the participants to rate their agreement with these various test names.

4.1 Subjects

We recruited participants by publishing the link to the survey on various platforms (Twitter, LinkedIn, etc.). After agreeing to data collection and participating in the study, the participants were asked 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.

16 participants took part in the study.

4.2 Treatments

We considered three treatments: test names generated by the current TestCube implementation (`TestCube Implementation`), names generated by our method-coverage based approach (`NATIC`) and names written by experienced software developers/testers (`Expert`).

The experienced software developers/testers were a Computer Science bachelor student and a PhD student 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. Using this information the experts derived the manually written test names.

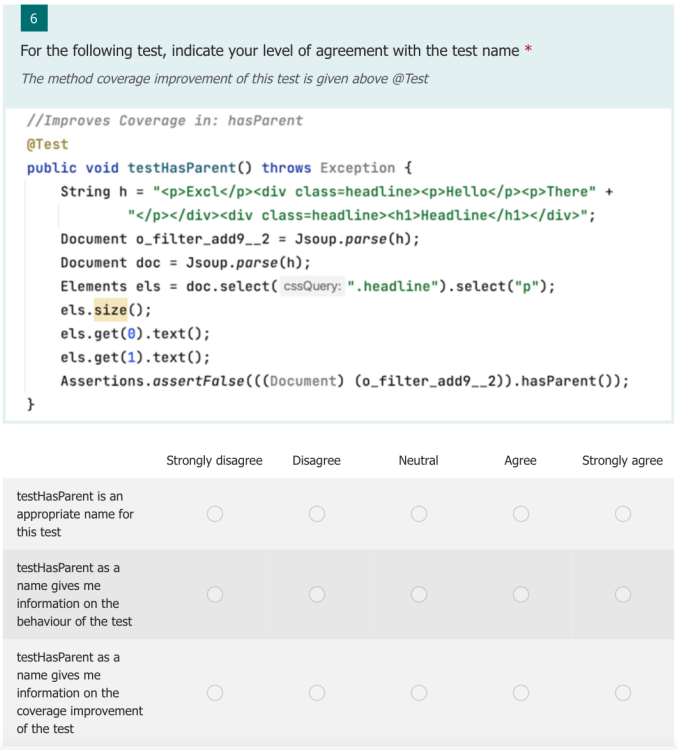

Figure 2: Example Question Research Study

4.3 Tasks

The participants were asked to rate their agreement for every amplified test case for these topics:

Appropriate Name: Given the original test case, the amplified test cases and their names, indicate your level of agreement with whether the name is an appropriate name for the test.

Behaviour: Given the original test case, the amplified test cases and their names, indicate your level of agreement with whether the name gives you information on the behaviour of the test.

Coverage Improvement: Given the original test case, the amplified test cases and their names, indicate your level of agreement with whether the name gives you information on the coverage improvement of the test.

4.4 Objects

We obtained the test names used in the study from running TestCube on the open-source HTML parser jsoup. We took the following steps to select and extract the test names:

- Download TestCube[5] and jsoup[6].
- Select 3 original unit test cases to be amplified. Any type of unit test that can be amplified in TestCube works for this research study.
- The result from the amplification consists of the amplified test cases with their names, `TestCube Implementation` or `NATIC`, and the test case's coverage improvement.
- For every amplified test case, all 3 types of names exist, and the survey randomly assigned tests to one type of name.

Table 1 gives examples of the names used in the study.

The `Expert` names were written by experienced Java developers. We added this metric because the `TestCube Implementation` names were trivial compared to the `NATIC` names, therefore the comparison would not have added a lot of insight for the study. The experienced Java developers had access to the original test case, the amplified test case and the coverage improvement, and derived test cases from this information.

4.5 Procedure

The study was performed as an online survey, which was hosted by Microsoft Forms. Every participant got the same sample of 25 test cases, which were amplified test cases from 3 original test cases. The participants were given the tasks from Section 4.3 (see example Figure 2), and at the final question, they could add feedback to either the survey or the test names. The order of the types of test names was randomized. With 25 test cases and 3 tasks, a total of 75 questions were answered for every participant in the study. The processed data leads to the distributions shown in Figures 3, 4, 5. For every test case, the participants were allowed to add one answer, in a 5-point Likert-scale (Strongly disagree, Disagree, Neutral, Agree, Strongly agree). The evaluation of each type of name is added to the y-axis of the chart in percentages.

4.6 Threats to Validity - Responsible Research

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 [14].

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

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

Internal Validity

Despite increasing the readability of the test cases by adding understandable names to them, the variable names were generated by TestCube. 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' to the survey, 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 [15], and the participants in our study all took at least the estimated 20 minutes to answer all the questions.

External Validity

The methods and tests from jsoup might not reflect larger, more complex test cases. However, TestCube 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.

5 Results

In this section, we illustrate the results from the research study we illustrated in Section 4. This sections contains the agreement percentages for the `TestCube Implementation`, `Expert`, and `NATIC` test names. The participants of the research study had the option to add free-text responses, which contains additional feedback.

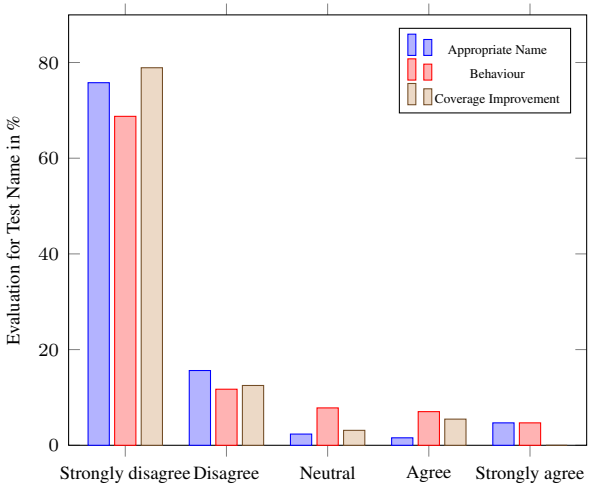

Figure 3: Distribution for `TestCube Implementation`

5.1 TestCube Implementation

The distribution in Figure 3 represents the results from the survey for the current TestCube generated test names. Overall answers for these test names resulted in a high level of (some) disagreement (either Disagree or Strongly disagree). For the appropriate name, the level of some disagreement was 91%, for information on the behaviour of the test the level of

(some) disagreement was 80%, and for information on coverage improvement it was 91%. This indicates that participants considered the test names not appropriate nor informative.

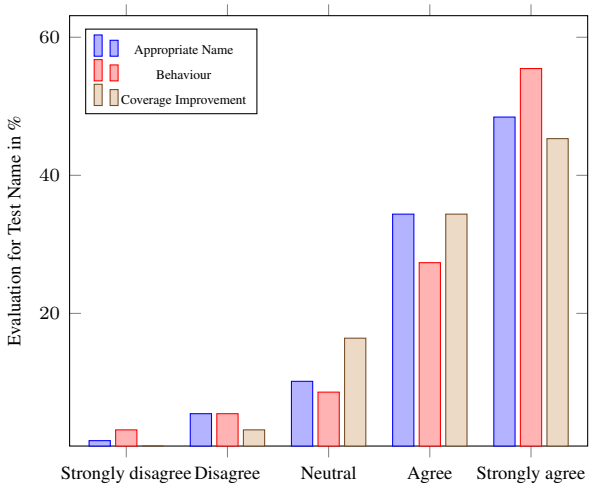

Figure 4: Distribution for `Expert`

5.2 Expert

The distribution in Figure 4 represents the results from the survey for the test names manually written by experts. The agreement with these names is significantly higher than the TestCube implemented 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 behaviour 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.

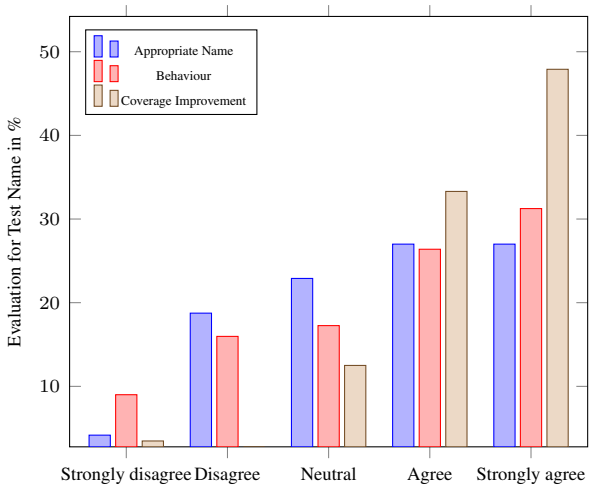

Figure 5: Distribution for `NATIC`

5.3 NATIC

The distribution in Figure 5 represents the results from the survey for the test names generated by `NATIC`. The two

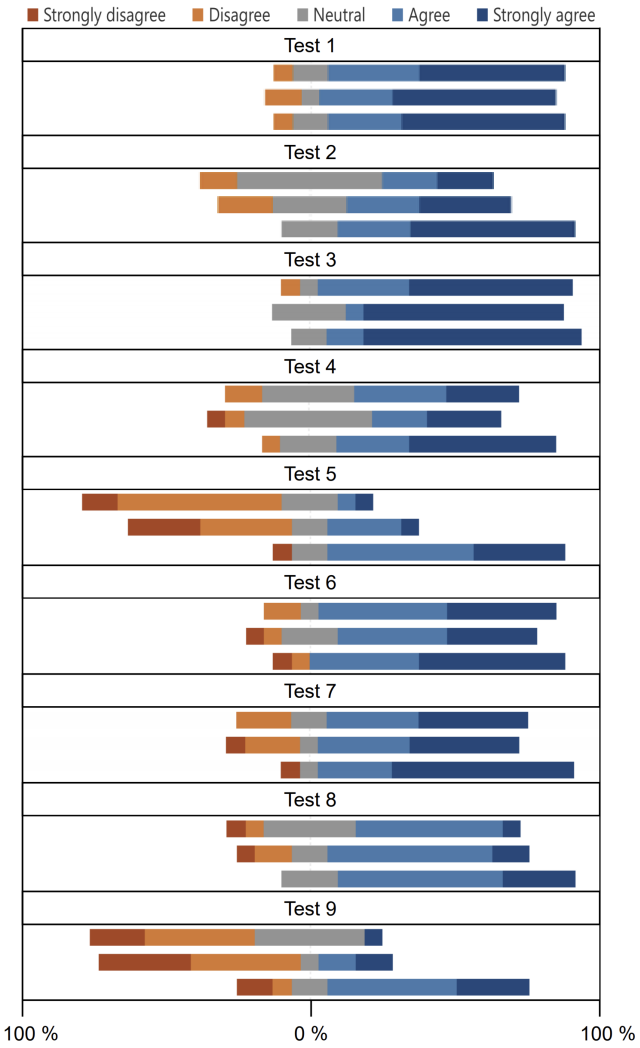

Figure 6: Likert agreement for each of the tests from `NATIC`

previous distributions showed a clear (dis)agreement distribution. The results for `NATIC` were slightly more spread. Since the preference for test names from this approach was more spread, a breakdown per test can be found in Figure 6.

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

5.4 Free-text Responses

All participants could give some 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 additional 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 additional input parameters, like the input to the test or the action taken by the test.

Answer to RQ2

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 some additional constraints are effective at generating appropriate names and give information on the amplified test case.

6 Discussion

From the individual results per test in Figure 6, 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. From the results of the other test names, we may assume that participants liked the shorter names better. Test 1 and Test 3 (see Listing 2) had particularly high agreement rates, which aligns with the assumption that more than 2 goals negatively impacts the agreement with the test names.

```
// Test 1
// Improves Coverage in: hasParent
@Test
public void testHasParent() throws Exception {}

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

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

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

Listing 2: Test Names Generated by `NATIC`

The participants of the study collectively disagreed with the names given by the `TestCube Implementation`. The `Expert` implementation had a higher agreement rating than `NATIC`.

Based on the results from Section 5.3 and the distribution shown in Figure 6 we formulate the following hypothesis: The test names generated by `NATIC` should contain at most 2 coverage goals. Future research has to be done to confirm this hypothesis.

7 Conclusions and Future Work

Amplified test names generated by TestCube need understandable and descriptive names. Benefits of "good" names are the ease of identifying the functionality the test checks and documenting the class under test.

In this paper, we have presented an approach, `NATIC`, which is based on the Daka 2017 approach [4], however alters the selection of goals and only takes covered methods as goals. `NATIC` uses the test suite generated by TestCube from an original test case as input, and the names of the methods in which the coverage is improved as coverage goals. `NATIC` generates a unique test name for every amplified test case in the suite.

The research study showed that the generated names were a great improvement compared to the current naming from TestCube, and with additional constraints on the amount of goals in the names, could compare to manually written test names.

Currently `NATIC` only generates names for amplified test cases. To further improve the readability of the body of the test cases, we would add descriptive variable names to the amplified test cases.

References

[1] C. Brandt and A. Zaidman, "Developer-friendly test amplification," 2021.

[2] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, *Modeling Readability to Improve Unit Tests*. ESEC/FSE 2015, New York, NY, USA: Association for Computing Machinery, 2015.

[3] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 625–636, 2016.

[4] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, (New York, NY, USA), p. 57–67, Association for Computing Machinery, 2017.

[5] 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 *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, (New York,

NY, USA), p. 287–298, Association for Computing Machinery, 2020.

[6] G. Grano, C. De Iaco, F. Palomba, and H. C. Gall, "Pizza versus pinsa: On the perception and measurability of unit test code quality," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 336–347, 2020.

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

[8] 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 *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 59–68, 2006.

[9] 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*, pp. 231–255, Springer, 2002.

[10] 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, (New York, NY, USA), p. 416–419, Association for Computing Machinery, 2011.

[11] L. Baresi and M. Miraz, "Testful: automatic unit-test generation for java classes," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, pp. 281–284, 2010.

[12] 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.

[13] B. Zhang, E. Hill, and J. Clause, "Automatically generating test templates from test names (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 506–511, 2015.

[14] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 666–676, 2015.

[15] 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.