DeepTC-Enhancer: Improving the Readability of Automatically Generated Tests

Devjeet Roy
devjeet.roy@wsu.edu
Washington State University

Ziyi Zhang
ziyi.zhang2@wsu.edu
Washington State University

Maggie Ma
yuzhanm@amazon.com
Infra Supply Chain & Automation, Amazon

Venera Arnaoudova
venera.arnaoudova@wsu.edu
Washington State University

Annibale Panichella
a.panichella@tudelft.nl
Delft University of Technology

Sebastiano Panichella
panc@zhaw.ch
Zurich University of Applied Science

Danielle Gonzalez
dng2551@rit.edu
Rochester Institute of Technology

Mehdi Mirakhorli
mehdi@se.rit.edu
Rochester Institute of Technology

ABSTRACT

Automated test case generation tools have been successfully proposed to reduce the amount of human and infrastructure resources required to write and run test cases. However, recent studies demonstrate that the readability of generated tests is very limited due to (i) uninformative identifiers and (ii) lack of proper documentation. Prior studies proposed techniques to improve test readability by either generating natural language summaries or meaningful methods names. While these approaches are shown to improve test readability, they are also affected by two limitations: (1) generated summaries are often perceived as too verbose and redundant by developers, and (2) readable tests require both proper method names but also meaningful identifiers (within-method readability).

In this work, we combine template based methods and Deep Learning (DL) approaches to automatically generate test case scenarios (elicited from natural language patterns of test case statements) as well as to train DL models on path-based representations of source code to generate meaningful identifier names. Our approach, called `DeepTC-Enhancer`, recommends documentation and identifier names with the ultimate goal of enhancing readability of automatically generated test cases.

An empirical evaluation with 36 external and internal developers shows that (1) `DeepTC-Enhancer` outperforms significantly the baseline approach for generating summaries and performs equally with the baseline approach for test case renaming, (2) the transformation proposed by `DeepTC-Enhancer` results in a significant increase in readability of automatically generated test cases, and (3) there is a significant difference in the feature preferences between external and internal developers.

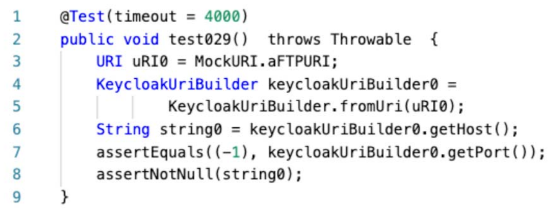
ACM Reference Format:
Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. `DeepTC-Enhancer`: Improving the Readability of Automatically Generated Tests. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416622

1 INTRODUCTION

Software testing is a crucial part of the software development life cycle that ensures software system quality and reliability properties [9]. However, writing tests is a resource-intensive endeavor; developers often spend 25% of their development time on software testing [11]. To help reduce the cost of testing, software engineering researchers have developed several approaches to generate tests automatically. A significant amount of progress has been made primarily in the case of unit tests. Today, there exist several tools, such as EvoSuite [16] and Randoop [35], that can automatically generate an entire test suite given a project's source code (or byte-code). The maturity of the field also resulted in surveys [10, 31] and several editions of tool competitions [15, 25, 32]. Furthermore, empirical studies showed that the tests synthesized by these tools are effective [5, 18] at detecting faults, and are substantially cheaper to produce [37].

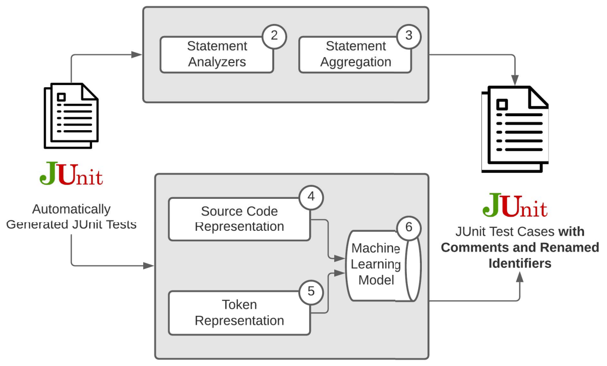
Despite these advances, generated unit tests pose a significant maintenance burden when including them in a project [12]. This is because developers have to manually validate the generate assertions (oracle problem) and analyze the thrown exceptions [24] (potential crashes). These automatically written tests have poor readability compared to their human-written counterparts due to the lack of documentation and the use of obfuscated variable names. Consider for example, the test case in Figure 1, which is automatically generated using EvoSuite [16] for the class `KeycloakUriBuilder` from the Keycloak open-source project. While the test method is concise, its purpose is not immediately obvious. The variable names have no clear purpose and just tell us the types of the instantiated objects and primitive types and their counts. Besides, the name of the test in itself is generic, and there are no comments to provide any hints about the scenario under test.

In recent years, researchers have proposed various approaches to partially mitigate these issues. The related work can be classified into two main categories: (1) generating natural language summaries to support comprehension, and (2) improving the test code for better readability. Panichella et al. [38] proposed a template-based summary generator for automatically generated tests. Their empirical study showed that test summaries help developers during debugging, i.e., finding more bugs and in less time. Daka et al. [14]



```java
@Test(timeout = 4000)
public void test029()  throws Throwable  {
    URI uRI0 = MockURI.aFTPURI;
    KeycloakUriBuilder keycloakUriBuilder0 =
            KeycloakUriBuilder.fromUri(uRI0);
    String string0 = keycloakUriBuilder0.getHost();
    assertEquals((-1), keycloakUriBuilder0.getPort());
    assertNotNull(string0);
}
```

Figure 1: A unit test generated by EvoSuite.



Statement Analyzers
2
Statement Aggregation
3
JUnit
Automatically Generated JUnit Tests
Source Code Representation
4
Machine Learning Model
6
Token Representation
5
JUnit
JUnit Test Cases with Comments and Renamed Identifiers

Figure 2: Overview of DeepTC-Enhancer.

proposed a technique to generate descriptive test names based on code coverage heuristics. Their empirical study showed that humans perceive the synthesized names as descriptive as manually-written test names [14]. While these two approaches address the test readability problem in different and complementary ways, they both have significant limitations. First, developers often perceive the generated summaries as too detailed (statement-level comments) and redundant [38]; they also do not solve the nondescript naming convention (developers' feedback reported in [38]). Furthermore, the approach from Daka et al. [14] considers only test method names, while the actual content of the test methods remains unchanged. In other words, the readability of the generated tests is still affected by meaningless identifier names (e.g., `string0` in Figure 1).

To address these open challenges, we propose a two-stage approach, called `DeepTC-Enhancer`, that comprehensively improves the readability of automatically generated unit tests. First, `DeepTC-Enhancer` automatically generates *test case scenarios* using a template based approach. These scenarios are method-level summaries, i.e., leading comments, that aim to summarize the steps, i.e., the scenario being set up and tested by a given test case. Our test scenarios differ from those generated by existing approaches [38] in the level of abstraction—they are more high-level (method-level as opposed to statement-level) and, therefore, more concise. The rationale is that higher-level summaries will quickly provide developers with enough information to decide whether the given test case needs to be further investigated for the task at hand. Second, `DeepTC-Enhancer` relies on extreme code summarization techniques based on Deep Learning to rename all identifiers in the test case with meaningful names. We hypothesize that such renaming can significantly increase the readability of these test cases and ease program comprehension and test maintenance activities.

Specifically, our contributions can be summarized as follows:

- A novel approach for generating natural language scenarios of JUnit test cases. `DeepTC-Enhancer` generates test method-level summaries that describe the test case scenarios.
- A novel adaptation of an existing identifier renaming technique applied in the context of unit tests in Java and adapted to remove its reliance on existing identifier names.
- An empirical evaluation of `DeepTC-Enhancer` using 6 internal and 30 external developers, including a comparison to existing approaches [14, 38].
- A replication package[1] that includes (1) a prototype implementation of the proposed approach, (2) internal and external developer surveys and (3) the data used for the evaluation.

Paper Structure. Section 2 details of the implementation of `DeepTC-Enhancer`. Section 3 provides an overview of the study design and research questions. Section 4 discusses the results, while threats to its validity are discussed in Section 5. Section 6 provides an overview of the related work and contrasts the work proposed here to the state-of-the-art approaches. Finally, Section 7 concludes the paper and outlines directions for the future work.

2 THE DEEPTC-ENHANCER APPROACH

Figure 2 depicts the proposed `DeepTC-Enhancer`, which is designed to automatically generate method-level summaries and automatically rename identifiers contained within leveraging (i) existing approaches on code summarization and (ii) deep learning techniques. In this section, we elaborate these steps, detailing `DeepTC-Enhancer`'s approach, which consists of two main phases: (1) Test Case Scenario-based Summaries and (2) Test Case Identifier Renaming.

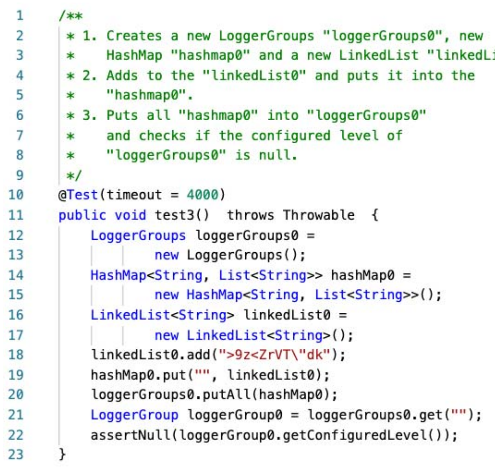
2.1 Test Case Scenario-based Summaries

At a high level, the summary generation phase starts by analyzing each line of code to filter out redundant information. Then, the remaining statements are aggregated to allow the actual generation of the method-level summary. Following is an in-depth description of each step in the summary generation phase.

2.1.1 Statement Analysis. We analyze all statements in the generated test case, looking for opportunities to reduce redundancy. For this, we employ intra- and inter-statement heuristics. *Intra-statement* heuristics determine whether details can be removed from within a single statement. For example, if a statement contains a method call with more than one argument, we remove the details about the parameters of the method call. *Inter-statement* heuristics determine which statements should be summarized. For example, if a temporary variable is created in a test case as a placeholder, then its creation does not need to be summarized as shown

[1]https://github.com/devjeetr/DeepTC-Enhancer-Improving-the-Readability-of-Automatically-Generated-Tests

in Figure 3 where the variable in line 21 is not part of the summary. Another example, also shown in Figure 3, is when multiple variable declarations (l.12–17) can be summarised in one sentence (l.2–3). The rationale here is that providing detailed information about every statement brings the summary too close to the actual source code. Instead, we hypothesise that if developers need more information, they will refer to the source code to procure it. Thus, the summaries that `DeepTC-Enhancer` generates are intended to guide developers in identifying relevant tests for their maintenance activity, rather than to convey detailed information about the tests.



```
1   /**
2    * 1. Creates a new LoggerGroups "loggerGroups0", new
3    *    HashMap "hashmap0" and a new LinkedList "linkedList0".
4    * 2. Adds to the "linkedList0" and puts it into the
5    *    "hashmap0".
6    * 3. Puts all "hashmap0" into "loggerGroups0"
7    *    and checks if the configured level of
8    *    "loggerGroups0" is null.
9    */
10  @Test(timeout = 4000)
11  public void test3()  throws Throwable  {
12      LoggerGroups loggerGroups0 =
13              new LoggerGroups();
14      HashMap<String, List<String>> hashMap0 =
15              new HashMap<String, List<String>>();
16      LinkedList<String> linkedList0 =
17              new LinkedList<String>();
18      linkedList0.add(">9z<ZrVT\"dk");
19      hashMap0.put("", linkedList0);
20      loggerGroups0.putAll(hashMap0);
21      LoggerGroup loggerGroup0 = loggerGroups0.get("");
22      assertNull(loggerGroup0.getConfiguredLevel());
23  }
```

Figure 3: Example of test case scenario-based summary generated by `DeepTC-Enhancer`.

2.1.2 Statement Aggregation. Once redundancy has been reduced, `DeepTC-Enhancer` aggregates the remaining statements into a final summary. To this end, `DeepTC-Enhancer` takes a template-based approach as templates have been shown to be successful for the generation of summaries [28, 38, 41]. `DeepTC-Enhancer` uses a set of heuristics that we crafted by manually investigating 293 JUnit tests files from over 31 open-source projects. During this manual investigation process, we identified several common sequences of statements that appear in unit tests and that can be summarized by a single statement. For example, a common pattern is the use of several assertions verifying different properties of a target object as shown in Figure 1. In this example, the last two assertions (l.13–14) can be aggregated into a single phrase "checks if port is -1 and host is not null". Another common pattern is a method invocation and an assert statement on the same object as shown in Figure 3 (l.20 and 22) which are summarized as one step (step 3) in the test case scenario. The comment in Figure 3 represents the summary generated by `DeepTC-Enhancer`.
The aggregation is performed iteratively, resulting in aggregated statements themselves being combined with others when applicable. We use a simple abstraction to enable this behavior: each individual or aggregated statement is assigned an *object* and an *action.* When test case statements do not fall under these aggregation patterns, we simply provide the statement level scenarios. Our templates cover 97% of the statements in the automatically generated tests from the projects described in Section 3.3.1.[2].

2.2 Identifier Renaming

For the identifier renaming phase of the proposed approach, we leverage and adapt existing deep learning approaches for extreme source code summarizations [2, 4, 6, 8]. The identifier renaming process consists of two separate prediction tasks: test case renaming and variable renaming. There exist several techniques that have achieved substantial success in both of these tasks [4, 6, 40]. Recently, deep learning techniques have achieved state-of-the-art performance for predicting the method name from the body of a method. Conversely, structured prediction has been successfully applied to the task of clarifying variable names from the obfuscated code [40]. Motivated by the success of machine learning models for these well known tasks, we divide the identifier renaming aspect of our approach into two learning tasks: test case name prediction and variable name prediction. We train the model on open source projects, as detailed in Section 2.2.4, and then use it to predict test case and variable names for automatically written tests. The rationale is that if the model can learn to predict identifier names used in human written test cases, it would be able to predict similar names for automatically written tests.

One key difference between this and prior works for these tasks is that our model does not rely on variable names already presented in the source code. This is important as our approach is being applied to automatically generated test cases that lack meaningful variable names. For both the variable and the test case prediction tasks, this is done by masking all variable names during the training of the model; we mask the variable that needs to be renamed with a special token.

2.2.1 Source Code Representation. One of the most important considerations when designing a machine learning system for software systems is source code representation. In practice, this can vary from simplistic representations of source code as a stream of tokens [21] to more structured, graph-based representations [3]. Our rationale for the selection process is two-fold:

(1) Cost: The cost of generating the representation should be minimal, in order to incorporate this tool as an IDE plugin.
(2) Dependencies: The source code representation must be generated solely from the raw text of the given test case. We impose this constraint to minimize the configuration burden on the developer end.

Based on these criteria, we use a path-based representation which Alon et al. proposed [7] and applied for various source code summarization tasks [6, 8]. In this approach, a section of source code is represented as an unordered set of *abstract syntax tree (AST) paths.* Each path represents a walk between two leaves in the AST of a program. We direct readers to [6–8] for detailed treatments and formalized definitions.

At the time it was proposed, this path-based representation produced state-of-the-art performance for several summarization tasks.

[2]Details regarding the aggregations and their associated templates can be found in the replication package

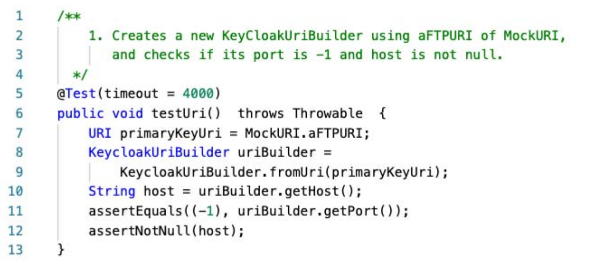
Since then, other approaches have been proposed that utilize alternative abstract syntax tree representations [27] or use a hybrid approach that combines an AST representation with a textual representation [44], with varying levels of performance. We chose the path-based representation because it is very cheap to compute and offers a level of generalization that enables it to be applied unmodified to our two learning tasks: variable name prediction and test case name prediction.

2.2.2 *Machine Learning Model.* Many of the advances made in code summarization over the past 5 years frame the problem as a translation problem, i.e., translating source code to natural language. This formulation allows for the use of a vast variety of approaches dealing with sequence transduction developed for neural machine translation (NMT) systems. For both of our learning tasks, we adapt the model proposed by Alon et al. [6]. This model follows the standard encoder-decoder architecture that, until recently, has been primarily utilized in NMT systems. The main difference between the model proposed by Alon et al. and the standard encoder-decoder architecture is that it is designed to use the path based representation discussed in the previous section. The model uses a specialized encoder to create a distributed representation of each AST path, which is then merged with token embeddings for the two terminal nodes of the path. This is then used by the decoder to sequentially generate the target prediction.

2.2.3 *Token Representation.* Originally, code2seq utilized subtoken embeddings to represent source code tokens. In this approach, tokens in the source code are split into subtokens. Recently, Karampatsis et al. show that the use of subword embeddings such as byte pair encoding (BPE) can significantly decrease the size of the vocabulary and improve the performance of machine learning models in the context of source code tasks [23]. Hence, for this work, we utilize SentencePiece BPE [26] based vocabulary. For the label (method or variable names) subtokens, we limit the vocabulary size to 16,000, and for the terminal node subtokens (any identifier that is not a variable or method name), we limit the vocabulary to 32,000.

2.2.4 *Dataset.* Deep learning approaches are typically data inefficient, requiring a lot of training data. However, we must ensure that our predicted identifier names would be of high quality. Manual validation of the dataset is not feasible, due to the scale of data we needed to collect. To ensure only *engineered* software projects are considered for this work, we used the dataset Munaiah et al. generated using `REAPER` [34]. This dataset includes quality metrics for each project. We used this to filter dataset for projects with a test-to-source code ratio greater than 0.01. We selected this threshold using descriptive statistics to find a compromise between the quality of the unit tests and the size of the resulting dataset. After filtering, we extracted all Java files that start or end with "Test", which is a common naming convention for unit test files. The final dataset consists of 274 engineered projects containing 96,534 unit test files for a total of 678,860 unit test cases. We divided the dataset into training (70%), validation (10%), and test set (20%) for our model.

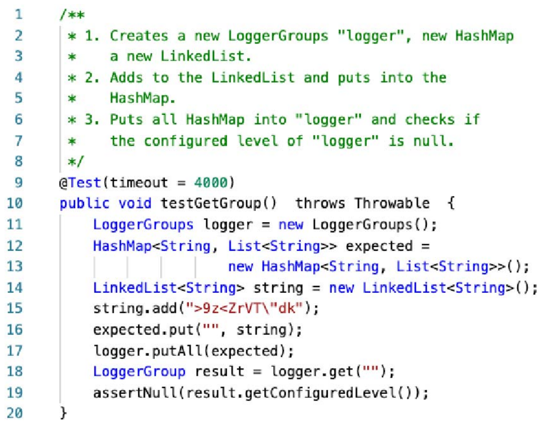
2.2.5 *Examples of suggested names.* For the automatically generated test shown in Figure 1, `DeepTC-Enhancer` suggests `testUri` as test name and `primaryKeyUri`, `uriBuilder`, and `host` for variables `uRI0` (l.3), `keycloakUriBuilder0` (l.4), and `string0` (l.6), respectively.



```java
/**
   1. Creates a new KeyCloakUriBuilder using aFTPURI of MockURI,
      and checks if its port is -1 and host is not null.
  */
@Test(timeout = 4000)
public void testUri()  throws Throwable  {
    URI primaryKeyUri = MockURI.aFTPURI;
    KeycloakUriBuilder uriBuilder =
        KeycloakUriBuilder.fromUri(primaryKeyUri);
    String host = uriBuilder.getHost();
    assertEquals((-1), uriBuilder.getPort());
    assertNotNull(host);
}
```

Figure 4: Test case from Figure 1 enhanced using `DeepTC-Enhancer`.

For the example shown in Figure 3, `DeepTC-Enhancer` suggests the test to be renamed to `testGetGroup` and variables `loggerGroups0` (l.12), `hashMap0` (l.14), `linkedList0` (l.16), and `loggerGroup0` (l.21) to be renamed to `logger`, `expected`, `string`, and `result`, respectively. Figure 4 and Figure 5 show the test cases from Figure 1 and Figure 3 with suggested method name and variable names generated by `DeepTC-Enhancer`.



```java
/**
 * 1. Creates a new LoggerGroups "logger", new HashMap
 *    a new LinkedList.
 * 2. Adds to the LinkedList and puts into the
 *    HashMap.
 * 3. Puts all HashMap into "logger" and checks if
 *    the configured level of "logger" is null.
 */
@Test(timeout = 4000)
public void testGetGroup()  throws Throwable  {
    LoggerGroups logger = new LoggerGroups();
    HashMap<String, List<String>> expected =
                    new HashMap<String, List<String>>();
    LinkedList<String> string = new LinkedList<String>();
    string.add(">9z<ZrVT\"dk");
    expected.put("", string);
    logger.putAll(expected);
    LoggerGroup result = logger.get("");
    assertNull(result.getConfiguredLevel());
}
```

Figure 5: Test case from Figure 3 enhanced using `DeepTC-Enhancer`.

3 STUDY DEFINITION AND DESIGN

3.1 Research Questions

The *goal* of this study is to evaluate the ability of `DeepTC-Enhancer` to improve the readability of automatically generated test cases using test case scenarios and identifier renaming. The *quality focus* is the evaluation of tool's performance from the perspective of developers. The *perspective* of the study is developers who are interested in using automatically generated test cases but struggle with their readability. Hence, the study is designed to answer the following research questions (RQs):

RQ$_1$: *How does* `DeepTC-Enhancer`*perform compared to existing techniques?* Several other approaches aim to improve the readability of automatically generated tests, by creating test case summaries *or* by providing more meaningful test names. As `DeepTC-Enhancer` enhances *both* the documentation (by adding method-level summaries) and code (by renaming identifiers) of generated tests, we compare its performance against the existing approaches for that generate both. For this RQ, we use the datasets used in the original evaluations of existing approaches to survey external developers.

RQ$_2$: *To what extent does* DeepTC-Enhancer *increase the readability of the generated tests?* Automatic test case generation requires fewer resources than the standard manual efforts. However, they incur a higher maintenance effort due to their poor readability. For this RQ, in addition to external developers, we also perform an evaluation with internal developers to gain insight on how useful developers might find DeepTC-Enhancer in their day to day work. For this RQ, we use a dataset that we collected consisting of top starred Java projects on GitHub.

RQ$_3$: *What aspects of* DeepTC-Enhancer *do developers find most useful?* We aim to assess whether developers perceive some enhancements applied by our approach more useful than others. For this RQ, we use the same developer pool and dataset as in RQ$_2$.

3.2 Baselines

To the best of our knowledge, DeepTC-Enhancer is the first approach to apply *both* automated documentation and test code enhancement (via identifier renaming) towards improving the readability of automatically generated test cases. However, there are approaches that improved individually one or the other aspects. Hence, we compare the test scenarios generated by DeepTC-Enhancer with the test case summaries generated by TestDescriber, proposed by Panichella et al. [38]. TestDescriber automatically generates test case summaries of the portion of code exercised by each test to provide a dynamic view of the class under test. The generated summaries have been shown to help developers to better understand the code under test and improve their bug fixing performance [38]. We compare the test case names generated by DeepTC-Enhancer with the names generated by Daka et al.'s approach [14]. The later synthesizes descriptive names for automatically generated unit tests in terms of their observable behavior at a test code level. This technique has been implemented as an extension to EvoSuite.

3.3 Experiment Design

To answer our research questions, we conduct two online empirical studies involving internal and external developers. Section 3.3.2 provides detailed information about the participants and their programming experience. We recruited participants via e-mail and social media. For each internal developer, we create a specific version of the survey for the open-source project for which they are listed as a contributor. The use of an online questionnaire was preferred over in-person interviews, as it is more convenient for the participants. Each survey starts by gathering demographic information. Then participants are asked to respond to a series of questions regarding the generated summaries, method renaming, and variable renaming. The survey also contained open-ended feedback forms after each question to allow the developers to provide additional insight into their responses.

3.3.1 Objects. We use three different datasets to evaluate different aspects of DeepTC-Enhancer:

1) Test summaries: We used the dataset used by Panichella et al. [38] to compare the test case summaries generated by DeepTC-Enhancer and TestDescriber in the survey with external developers. The dataset consists of two Java classes extracted from two open-source projects. We use this dataset to answer RQ$_1$, RQ$_2$, and RQ$_3$.

2) Method renaming: Daka et al. [14] followed a systematic protocol [17] to select objects from the SF110 corpus[3] of open-source Java projects. They selected ten target methods from different classes. We use data from this dataset to answer RQ$_1$, RQ$_2$, and RQ$_3$.

3) Variable renaming and evaluation of the overall approach: We use the 30 most-starred open-source Java projects from GitHub. We use the same dataset to recruit internal developers for our evaluation. This dataset was used to evaluate the quality of the suggested variable names and the overall approach that includes the generated summaries, method names, and variable names. It is used to answer RQ$_2$ and RQ$_3$.

Table 1: Experience of Participants.

	External	Internal
Experience	# (%)	#
0-2 years	6 (20%)	0
3-6 years	14 (47%)	1
7-10 years	7 (23%)	0
>10 years	3 (10%)	5
Total	30 (100%)	6

3.3.2 Participants. We recruited both external developers, i.e., people that have not developed the code under investigation, and internal developers, i.e., contributors of the projects we are analyzing. We invited external developers from industry, students, and researchers from the authors' institutions as well as from other institutions by e-mail and social media. For our study with internal developers we invited 198 top contributors from the 30 most starred open-source Java projects from GitHub. At the end, 30 external and 6 internal developers responded to our surveys. Table 1 details the participants' programming experience. All participants have a Computer Science background. They were all volunteers and did not receive any reward for participation in the study.

3.3.3 Surveys. We performed two different surveys: one with external and one with internal developers. Here we briefly describe the surveys; more details can be found in our replication package.

1) External developers survey: The purpose of this survey is to evaluate several research tools developed to enhance the readability of generated tests. We create two versions of the survey to contain different code snippets which are randomly selected from the corresponding datasets. Participants are randomly assigned to a survey and asked to evaluate the enhancements using different criteria. The survey consists of 17 questions divided into four sections; there is also optional feedback forms intended to allow participants to elaborate on their answers. In Section 1, participants are asked to evaluate the quality of the test case summary generated by our approach and the baseline TestDescriber [38]. In Section 2, they evaluate the quality of the test names suggested by our tool and by the work of Daka et al. [14]. Sections 3 contains an evaluation of the variable renaming in isolation, and Section 4 contains an evaluation of the overall approach. To minimize order and sequence

[3]http://www.evosuite.org/experimental-data/sf110/

effects as well as other bias, for Sections 1 and 2, the identity of the tools is not revealed, and the order in which the summaries and method names are presented is randomized.

2) Internal developers survey: The goal of this survey is to evaluate the usefulness of `DeepTC-Enhancer` for software developers. For all developers that indicate an interest in participating, we create a unique survey, even if there is more than one developer for a particular project. The survey consists of 16 questions and optional feedback forms divided into four sections. Three of these sections evaluate the test case scenario, method renaming, and variable renaming in isolation, while the last section contains an evaluation for the complete approach. The same automatically generated test case is used for each of these sections.

3.4 Evaluation metrics

Similar to related work [33, 38, 42], the quality of generated summaries is evaluated according to three dimensions: conciseness, content, and readability. *Concise* summaries do not include extraneous or irrelevant information. *Content* measures whether the summary correctly reflects the content of the test case. *Readability* measures to what extent a test case (including the generated enhancements) is perceived as readable and understandable by the participants. In addition, participants evaluate how the *intent* of the test case are captured by the suggested test case name and variable names. Participants also rate the *naturalness* of the suggested test case name which indicates how easy it is to read and to understand it. Finally, participants are asked to rate the *improvement in readability* of the code snippet enhanced using `DeepTC-Enhancer` over the original automatically generated test and their likelihood to utilize `DeepTC-Enhancer` if they were to use automatically written tests. Depending on the question, participants express their opinions using a 3-, 4-, or 5-point Likert scale.

3.5 Analysis Method

We used statistical tests to assess the significance of the difference between the scores achieved by different tools. We use the Wilcoxon Rank Sum test with a significance level of $\alpha = 0.05$. We opted for non-parametric tests because the Shapiro-Wilk test revealed that our data (scores) does not follow a normal distribution (p-value < 0.01). Besides, we use Cliff's d effect size [20] to measure the magnitude of the difference, which can be interpretted as follows: *small* for $d < 0.33$, *medium* for $0.33 \leq d < 0.474$ and *large* for $d \geq 0.474$ [20].

4 RESULTS

4.1 RQ1 : *How does* `DeepTC-Enhancer` *perform compared to existing techniques?*

4.1.1 Test Case Summaries. For the reader to get a sense of the summaries generated by different tools, we present examples of summaries generated by `DeepTC-Enhancer` and TestDescriber in Figure 8 and Figure 7, respectively, for the same unit test. Figure 6 shows the results of the evaluation of the proposed test case scenarios when compared with TestDescriber [38]. In terms of *conciseness*, 19 (64%) respondents rated test case scenarios as containing no unnecessary information, while on the other hand, 11 (37%) rated TestDescriber's summaries as having no unnecessary information. For either approach, 7-9 respondents found that the summaries contain some unnecessary information. Only 3 respondents found test case scenarios to contain mostly unnecessary information, as compared to 3 (10%) respondents for TestDescriber summaries. In terms of *content adequacy*, test case scenarios scored better overall, with 17 (57%) respondents reporting them to contain all important information, while 7 (23%) and 4 (13%) respondents reported them to be missing some important information and missing some very important information, respectively. On the other hand, TestDescriber's

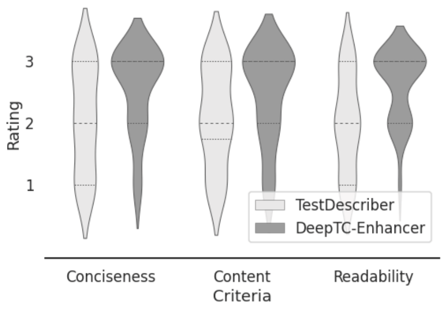
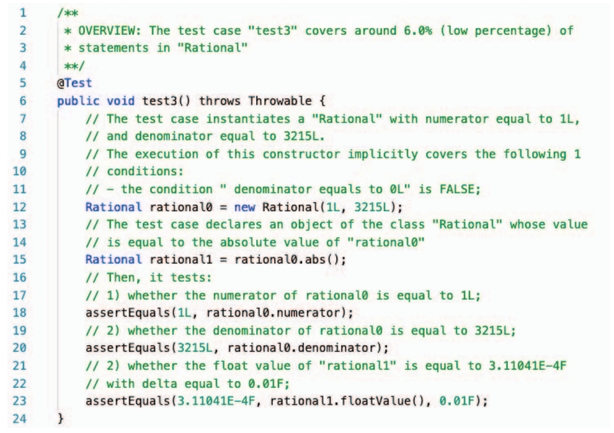
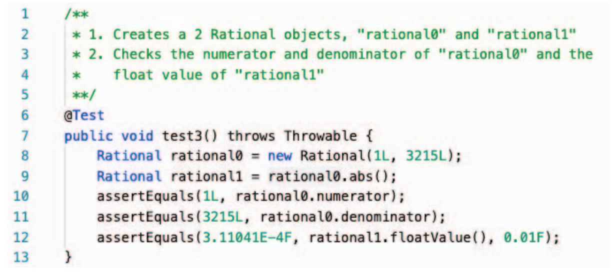


Figure 6: Results from the comparison of `DeepTC-Enhancer` and TestDescriber for method summaries on a 3-point Likert scale (higher numbers are better).



```
/**
 * OVERVIEW: The test case "test3" covers around 6.0% (low percentage) of
 * statements in "Rational"
 **/
@Test
public void test3() throws Throwable {
    // The test case instantiates a "Rational" with numerator equal to 1L,
    // and denominator equal to 3215L.
    // The execution of this constructor implicitly covers the following 1
    // conditions:
    // - the condition " denominator equals to 0L" is FALSE;
    Rational rational0 = new Rational(1L, 3215L);
    // The test case declares an object of the class "Rational" whose value
    // is equal to the absolute value of "rational0"
    Rational rational1 = rational0.abs();
    // Then, it tests:
    // 1) whether the numerator of rational0 is equal to 1L;
    assertEquals(1L, rational0.numerator);
    // 2) whether the denominator of rational0 is equal to 3215L;
    assertEquals(3215L, rational0.denominator);
    // 2) whether the float value of "rational1" is equal to 3.11041E-4F
    // with delta equal to 0.01F;
    assertEquals(3.11041E-4F, rational1.floatValue(), 0.01F);
}
```

Figure 7: Example summary generated by TestDescriber.



```
/**
 * 1. Creates a 2 Rational objects, "rational0" and "rational1"
 * 2. Checks the numerator and denominator of "rational0" and the
 *     float value of "rational1"
 **/
@Test
public void test3() throws Throwable {
    Rational rational0 = new Rational(1L, 3215L);
    Rational rational1 = rational0.abs();
    assertEquals(1L, rational0.numerator);
    assertEquals(3215L, rational0.denominator);
    assertEquals(3.11041E-4F, rational1.floatValue(), 0.01F);
}
```

Figure 8: Example summary generated by `DeepTC-Enhancer`.

summaries were reported to contain all important information by 10 (33%) respondents, missing some important information by 11 (37%) and missing some very important information by 7 (23%) respondents. Lastly, on the criteria of *readability*, most respondents found test case scenarios to be either easy (19 (63%)) or somewhat easy to read (9 (30%)), while only one respondent rated it as being difficult to read. TestDescriber's summaries were reported by 8 (26%) to be easy to read, somewhat easy to read by 12 (40%) and difficult to read by 8 (27%). Overall, 20 (67%) respondents preferred test case scenarios to TestDescriber's summaries, while 4 (13%) preferred neither.

From a statistical point of view, the results of the Wilcoxon test revealed that the test scenarios generated by DeepTC-Enhancer are perceived by participants as significantly more concise (p-value=0.02) and more readable (p-value≤0.01) compared to the summaries by TestDescriber. The effect size is *medium* in both cases, being 0.32 and 0.44, respectively. Instead, there is no significant difference in terms of quality of the *content* (p-value=0.07).

Discussion: Across all three criteria evaluated in the survey on external developers, test case scenarios performed better than TestDescriber's summaries. Based on our qualitative analysis of the respondents, we attribute these higher ratings to test case scenarios being shorter and less detailed than the summaries provided by TestDescriber. While DeepTC-Enhancer provides high-level summaroes of test cases at the method-level, the latter instead provides detailed, line by line summaries. Some respondents indicated that detailed comments is redundant, as reading the source code that follows each statement summary would provide them with the same information. This sentiment was echoed by another participant stated that the test case scenario was "concise, easy to read" whereas the for TestDescriber summary they could "tell from the code what it does". However, while overall DeepTC-Enhancer is received better than TestDescriber, two respondents did not see the value in the summaries.

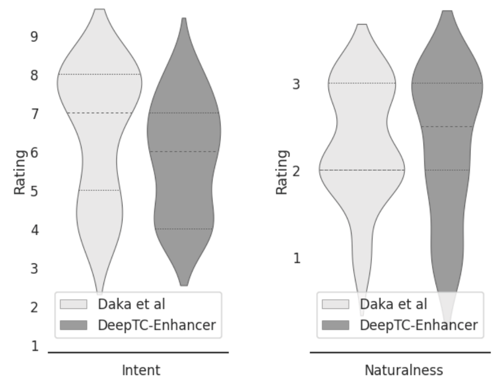
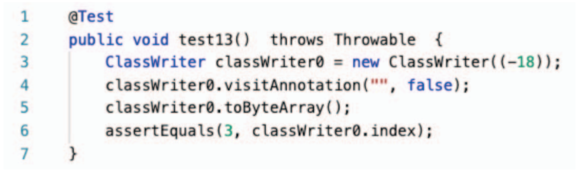


Figure 9: Results from the comparison of DeepTC-Enhancer and Daka et al. 's approach for test case names on a 5 and 3 points Likert scales for *intent* and *naturalness* respectively (higher numbers are better).



```
@Test
public void test13()  throws Throwable  {
    ClassWriter classWriter0 = new ClassWriter((-18));
    classWriter0.visitAnnotation("", false);
    classWriter0.toByteArray();
    assertEquals(3, classWriter0.index);
}
```

Figure 10: Example test case that needs renaming.

4.1.2 Test Case Names. Results for the comparison of Daka et al. 's method naming approach to DeepTC-Enhancer's method renaming are shown in Figure 9. More survey respondents found the method names provided by Daka et al. 's approach to fully capture the intent of the test case (13 (43%)) as compared to those suggested by DeepTC-Enhancer(4 (13%)). 7 (23%) respondents reported that the names suggested by DeepTC-Enhancer either mostly capture the intent of the test case as compared to 5 (17%) for Daka et al. 's approach. On the other end of the spectrum, 4 (13%) respondents reported the name generated by DeepTC-Enhancer as not capturing the intent of the test case while 3 (10%) reported the same for the baseline approach. A similar number of respondents (6 (20%) versus 8 (27%)) reported that the test case names proposed by either approach were misleading in conveying the intent of the test case. With regard to *readability*, the names suggested by DeepTC-Enhancer were perceived as slightly more readable than those suggested by the baseline. 14 (47%), 8 (27%), and 6 (20%) respondents reported the proposed approach as being easy, somewhat easy and difficult to read, respectively, while 11 (37%), 12 (47%) and 3 (10%) reported the same for Data et al. 's approach. Lastly, 11 (37%) respondents each indicated a preference for either tool, while 8 (26%) indicated they preferred neither.

However, the Wilcoxon test revealed that there is not significant difference in quality between the test names generated by DeepTC-Enhancer and the approach by Daka et al. [14]. More precisely, there is no significant difference in terms of *intent* (p-value=0.07) nor in terms of readability (p-value=0.84).

Discussion: Overall, Daka et al. 's approach performs slightly better in terms of capturing the intent of the test case, and slightly worse in terms of readability, albeit the differences are not statistically significant. Despite that, participants were equally likely to prefer names generated by either approach. We posit that this preference arose out of the conciseness of DeepTC-Enhancer's suggested names, and conversely the excessive verbosity of Daka et al. 's approach. Indeed, several respondents indicated that they prefer conciseness over verbosity for test case names, unless increased verbosity is required to address naming conflicts. For example, for the test case in Figure 10, Daka et al. 's approach suggests test name testVisitAnnotation-WithNonEmptyStringAndFalse and DeepTC-Enhancer suggests testVisitAnnotation. We also observe from the responses that names suggested by DeepTC-Enhancer might be perceived as too general. One respondent stated that this can be a problem when multiple test cases are similar to one another. Lastly, our results reveal that a significant portion (8, i.e., 26%) of the respondents did not prefer the method names from either tool. This indicates that the method names generated by both DeepTC-Enhancer and Daka et al. have quite some room for improvements. We surmise that respondents would prefer method names that combine some aspects of the verbosity of Dakaet al. 's names, while also retaining some of the conciseness of the names suggested by DeepTC-Enhancer.

RQ$_1$ External developers find the proposed test case scenarios generated by DeepTC-Enhancer to be more concise, content adequate, and readable than the summaries generated by the baseline. The difference is statistically significant for conciseness and readability with a medium effect size. They also found the method names suggested by DeepTC-Enhancer to be more concise than those of the baseline. In terms of readability and intent, the method names by DeepTC-Enhancer and the baseline are statistically equivalent.

4.2 RQ 2: *To what extent does* DeepTC-Enhancer*increase the readability of the generated tests?*

4.2.1 Overall Approach. We observe that respondents provided favorable ratings of the proposed approach's impact on the readability of automatically generated unit tests. 13 respondents (43%) reported that the enhancement applied by DeepTC-Enhancer resulted in a significant improvement in the readability of the generated test cases, while 11 (37%) reported that the increase in readability was minor. Three respondents (10%) indicated that there was no change in readability while 2 (7%) reported that there was a minor decrease in readability. No participant indicated there to be a significant decrease in readability as a result of the enhancements provided by the proposed approach. As to the likelihood of respondents using DeepTC-Enhancer to enhance automatically generated test cases, we found that 6 participants (20%) reported that they were extremely likely to use it, while 16 (53%) indicated that they were somewhat likely to use it. One respondent indicated that they were somewhat unlikely to use the proposed approach while two indicated that they were highly unlikely to use DeepTC-Enhancer. Lastly, most participants (20 (67%)) indicated that the existence of the proposed tool would make them more likely to utilize automated test generation tools in their projects. 7 participants (23%) indicated that the existence of the tool would have no effect on their choice of using automatic test generation tools; two respondents indicated that DeepTC-Enhancer was unlikely to have an effect on their choice to use these tools.

Internal developers, much like external developers, positively rated the impact of DeepTC-Enhancer on readability: 4/6 indicate a significant increase and 2/6 indicate a minor increase. However, they were not as likely as external developers to utilize DeepTC-Enhancer should they use automatic test generation; with 3/6 indicating they were highly or somewhat likely to use it, 1/6 neutral and 2/6 either highly or somewhat unlikely. Lastly, all 6 participants indicated that the presence of DeepTC-Enhancer would not make them more likely to use automatic test case generation. This is mostly because the generated tests do not match the code styles of their organizations.

4.2.2 Test Case Scenarios. We report the results of the evaluation of the proposed test case scenarios by internal developers in Table 2. Respondents rated the test case scenarios generated by

Table 2: Internal developers' feedback on test case scenarios generated by DeepTC-Enhancer.

Criteria	Rating	# Resp.
Conciseness	No unnecessary information	2/6
	Mostly unnecessary information	4/6
Content	Not missing important information	4/6
	Missing some very important information	2/6
Readability	Easy to read/understand	4/6
	Somewhat easy to read/understand	2/6

Table 3: External and Internal developer feedback on variable names suggested by DeepTC-Enhancer.

Rating	External	Internal
	# (%)	#
Fully conveys intent	167 (48%)	7/18
Somewhat conveys intent	121 (35%)	8/18
Does not convey intent	33 (10%)	2/18
Misleading with regard to intent	24 (7 %)	1/18

DeepTC-Enhancer favorably in terms of *readability* and *content adequacy*, but not in terms of *conciseness.* This is somewhat consistent with the external developer responses reported in Section 4.1.1. However, internal developers rated the conciseness of the scenarios much lower than the external counterparts. Their concerns mirrored those of external developers; 1 participant indicated that they preferred reading the actual source code instead of the summary comment to figure out what the test case exactly does. They also were more specific about code quality standards for unit tests; one developer indicated that a good test case should make it evident what is being tested; they would rather the intent of the test case be contained in the code than the comments. In general, internal developers were not in favor of have comments that describe the functionality of the test cases. Part of the reasons for that might be due to the fact that for the evaluation respondents were shown single test cases as opposed to the entire test suite generated by EvoSuite. We hypothesize that developers will see more value in the test case scenarios in the context of test case navigation. We plan to perform such evaluation as part of future work.

4.2.3 Test Case Names. Overall, internal developers found the method names suggested by DeepTC-Enhancer do not successfully capture the intent of the test cases presented to them. Four developers indicated that the suggested name either didn't capture intent (2) or was misleading (2). However, they rated the suggested names favorably in terms of readability, with 5 indicating that it was easy to read (3) or somewhat easy to read (2), with the remaining developer indicating the name to be not easy to read. This is once again similar to the results of the external developer survey reported in Section 4.1.2.

4.2.4 Variable Names. The survey results with external developers for the variable names suggested by DeepTC-Enhancer are shown in Table 3. Overall, participants reported that the names capture the intent of the variable usage to some extent 83% of the times, with the remaining 17% reporting that the variable renaming either did not

Table 4: External and internal developer feedback for usefulness of features

	Internal		External	
	Useful	Not Useful	Useful	Not Useful
Summary	3/6	2 / 6	20 (67%)	9 (30%)
Variable	5/6	0 / 6	23 (77%)	5 (17%)
Method	5/6	0 / 6	22 (73%)	6 (20%)

capture the intent or was misleading. 48% of the times, respondents indicated that the suggested variable name fully captured the intent of its usage.

Table 3 also shows the internal developer evaluation of the variable names on 18 different instances. Overall, they rated the variable names' ability to convey the intent of their usage fairly positively: seven variable names being rated as fully capturing the intent of their usage and eight being rated as somewhat capturing the intent of the usage. Two variable names were rated as not capturing the intent of their usage while one was rated is being misleading with regard to intent. In general, developers were very enthusiastic about the variable renaming feature. One developer indicated that the impact of the renaming of the variables would be much more apparent in larger test cases.

RQ₂ Overall, external and internal developers report that the transformations of `DeepTC-Enhancer` result in a significant increase in readability of automatic test cases. Internal developers are particularly enthusiastic about the variable renaming. Moreover, internal developers also find test case scenarios to be readable and content adequate, but lacking conciseness.

4.3 RQ3: *What aspects of* `DeepTC-Enhancer`*do developers find most useful?*

Table 4 shows the results from the evaluation with external developers. Respondents were asked to place the three features of `DeepTC-Enhancer` (scenarios, test case names, and variable names) in two buckets: useful and not useful. 20 (67%) respondents placed test case scenarios in the useful bucket, while nine (30%) placed them in the not useful bucket. Variable renaming were considered useful by 22 (73%) respondents and not useful by 6 (20%). Lastly, 23 (77%) respondents found the test case renaming to be useful, while 5 (17%) found it to not useful.

In terms of the relative importance of these three aspects of the proposed approach, the largest number of participants reported test case scenarios to be the most important feature, with 12 (40%) participants rating it as the most important, whereas 6 (20%) and 8 (27%) participants ranked variable and method renaming respectively as the most important feature. Test case scenarios were also the most frequently top-ranked feature rated by respondents (3) as not being useful.

The results for internal developers are in contrast with the results from external developers. 3/6 developers rank variable renaming as the feature they found the most useful while another ranked it second. Method name renaming is ranked as the second most useful feature by 3/6 developers. Lastly, internal developers did not find test case scenarios to be as useful as the external developers; in fact, 2/6 rated summaries as not being useful at all.

The results of the Wilcoxon test revealed that the test scenarios are significantly ranked higher than the other features in our tool in terms of usefulness as indicated in participants' answers (p-value=0.05). The effect size is *medium* (0.39) compared to variables names and *small* (0.283) compared to method names. Instead, there is no statistical difference between the other two features, i.e., , the usefulness (ranks) of methods and variables names.

Discussion: We observe that for external developers, test case scenarios are the most useful enhancement offered by `DeepTC-Enhancer`. This is contrary to our findings for internal developers. From a qualitative analysis of the comments left by respondents, we gather that internal developers often prefer self-documenting code over explicit documentation. While this is true even for external developers, the section of respondents that expressed this preference was in the minority as evidenced by the results. In addition, internal developers tended to perceive the test scenarios from the perspective of the project being evaluated; one of them indicated that both the coding style and the test scenarios of the automatically generated test presented to them did not fit their current coding quality standards. We also posit that the difference in the perception of test case scenarios for internal and external developers could be, at the very least, attributed to the familiarity the developers had with these projects. Some snippets used for internal developers were also used for the external evaluation, and given the external developers' lack of familiarity with the code, they found test case scenarios to be more helpful.

RQ₃ External developers consider the automatically generated test case scenarios as the most useful aspect of `DeepTC-Enhancer`, whereas internal developers prefer the variable renaming feature. We attribute the main reasons for the different opinions to the familiarity (or lack of it) with the code and the coding standards followed by different projects and developers.

5 THREATS TO VALIDITY

In this section, we outline possible threats to the validity of our study and show how we mitigated them.

Threats to construct validity concern the way in which we set up our study. Due to the fact that our study was performed in a remote setting in which participants could work on the tasks at their own discretion, we could not oversee their behaviours. To minimize potential bias in the participants' behaviours, we have shared the experimental data with the participants using an online survey platform, which support the participants (1) to perform tasks and (2) facilitate the filling of the questionnaires. In additional, to limit this threat, we also involved both external and internal developers, so that the final reported results are more reliable.

Threats to internal validity concern factors that might affect the casual relationship. To reduce biasing developers to the baselines evaluated, the name of the tools used to generate the summaries and identifiers names were not revealed in the survey. To avoid bias in the task assignment, we randomly assigned the tasks to the participants in order to have a balanced number of data points for all treatments. Specifically, for external developer surveys, participants were randomly assigned to one off the two surveys. For internal developers surveys, we created a unique survey, even if there was more than one developer for a particular project, we then randomly selected both test method and test class from the automatically generated test suite. Another factor that can influence our results is the order of assignments. However, our results suggest similar results among participants, thus, presenting no interaction between the treatments and the final outcome.

Threats to external validity concern the generalizability of our findings, and in particular on the evaluation of DeepTC-Enhancer. To limit this threat, we considered different dataset to evaluate our approach. To evaluate the quality of generated test summaries and suggested method names, we considered the original datasets used in the studies involving two baselines [14, 38]. We also use a new dataset containing 30 top starred projects on Github to evaluate DeepTC-Enhancer. We plan to evaluate DeepTC-Enhancer with a larger dataset with more complex test cases. We will also work on improving the suggestions for test case names. Future work will also focus on investigating how the approach helps developers fix potential bugs [38], which deserve future investigations. Finally, even if our population included a substantial part of professional internal and external developers, we plan to replicate this study with more participants in the future in order to increase the confidence in the generalizability of our results.

Threats to conclusion validity concern the degree to which our conclusions about DeepTC-Enhancer are reasonable based on the data. DeepTC-Enhancer generates test summaries and suggests identifier names for automatically generated test cases by Evosuite. Using different automatic test generation tools such as Randoop [35] might lead to different results. However, we observe that the test cases generated by Evosuite are not significantly different from those generated by other existing tools in terms of size, structure and coverage. We support our findings by using appropriate statistical tests, i.e. the non-parametric Wilcoxon test. We also used the Wilk-Shapiro normality test to verify whether the non-parametric test could be applied to our data. Finally, we used the Vargha and Delaney $\hat{A}_{12}$ statistical test to measure the magnitude of the differences between different approaches.

6 RELATED WORK

Source Code Summarization. Researchers have proposed several approaches that generate summarization of software artifacts at different levels of granularity to reduce program comprehension effort during software development and maintenance. At statement level, Gonzalez et al. develop an automated technique to convert JUnit assertion statements into natural language sentences [19]. At method level, Sridhara et al. [42] propose an approach that automatically generates natural language summary comments for Java methods. At class level, Moreno et al. [33] present an approach to generate human-readable summaries for Java classes so that developers can understand the main goal and structure of a class easily. McBurney and McMillan [30] propose an approach to generate automatic source code summaries with contextual meaning in them by analyzing how those methods are invoked to show why the method exists or what role it plays in the software. To locate crosscutting concern code, Rastkar et al. [39] introduce an automated approach that produces a natural language summary describing both what the concern is and how the concern is implemented so that developers can perform change tasks more efficiently. All these approaches focus on generating summaries for source code; we target automatically generated test cases and generate test scenario rather than statement-level descriptions as in [19].

Method and Variable Renaming. Allamanis et al. [2] introduce a neural probabilistic language model for source code that can suggest method names. In addition, Yonai et al. [45] propose an approach Mercem to recommend method names in source code by applying graph embedding techniques to the call graph. Both approaches target methods whereas DeepTC-Enhancer focuses on suggesting names for automatically generated test cases.

Raychev [40] build an engine called JSNICE to predict variable names and type annotations of JavaScript programs. In addition, Vasilescu et al. [43] presented an approach JSNAUGHTY to recover the original names from minified JavaScript variable names. However, both JSNICE and JSNAUGHTY are provided for JavaScript software system but not automatically generated Java test cases. To the best of our knowledge, no other work focus on enhancing the readability of variable names in automatically generated test cases in Java.

Improving the Readability of Test Cases. Kamimura and Murphy [22] propose generating human-oriented summaries of test cases based on static source code analysis. Li et al. [28] present an approach UnitTestScribe that combines static analysis, natural language processing, backward slicing, and code summarization techniques to generate descriptions documenting the purpose of methods within unit tests. UnitTestScribe works for C# project. Panichella et al. [38] introduce an approach, TestDescriber, to automatically generates test case summaries of the portion of code exercised by each individual test to improve. We compare our approach with the work of Panichella et al.

To generate descriptive method names for Java unit test cases, Zhang et al. [46] present an approach, NameAssist, that combines natural-language program analysis and text generation, which can create test method names that summarize the test's scenario and the expected outcome. We considered NameAssist as a candidate baseline but neither the dataset nor the tool is publicly available, hence we exclude it from this comparison. Daka et al. [14] introduce an approach for automatically generated tests that can generate descriptive names by summarizing API-level coverage goals. We use the approach by Daka et al. as a baseline for our comparison.

Afshan et al. [1] used a linguistic model to generate 'English-like" input strings, which are more understandable that randomly generated ones. Our work complements this line of research as we aim at improving readability in different dimensions, i.e., documentation and method/variable names. Our paper shows that these factors are perceived as very important by developers. Daka et al. [13] proposed a domain-specific model to measure and improve the readability of generated units test and based on human judgments. EvoSuite already incorporates heuristics to reduce the size of the generated tests and the number of assertions [16]. One could argue that, at the same level of coverage, shorter tests are easier to validate and inspect manually, reducing the oracle cost [36]. However, these studies focus on the structure, size, and complexity of the generated tests. Instead, we focus on natural language documentation (not included in the model by Daka et al. [13]) and the quality of the identifiers (that remain obfuscated).

As indicated by Lin et al. [29], the poor quality of the identifiers in test code is widespread, and it is more severe in generated tests compared to the manually-written counterpart. Our tool applies multiple strategies to address, among others, this open issue.

7 CONCLUSIONS AND FUTURE WORK

We propose DeepTC-Enhancer: a hybrid deep learning and template based approach to improve the readability of automatically generated test cases. DeepTC-Enhancer is the first approach that enhances both the documentation and code aspects of the tests. To this end, DeepTC-Enhancer generates leading tests case summaries in the form of test case scenarios which outline the actions performed in the test. To enhance the automatically generated identifiers of tests, DeepTC-Enhancer adapts a deep learning based extreme code summarization approach to generate test case names and variable names.

With two empirical evaluations involving 30 external and six internal developers, we evaluate all aspects of the transformations proposed by DeepTC-Enhancer and compare it to existing baselines. Results show that summaries generated using the proposed approach significantly outperform the baseline, and the test case renaming performs similar to the baseline. The variable renaming feature in the context of automatic tests is first of its kind and thus is not compared to a baseline. This is the feature that was preferred by internal developers, while external developers ranked the test case scenarios as a more important feature. Overall, both internal and external developers report a significant improvement in the readability of generated tests after the enhancement applied by DeepTC-Enhancer. Lastly, the majority of participants indicate that the existence of DeepTC-Enhancer increases the likelihood of using tools for automatic test case generation in their projects.

In the future, we plan to evaluate DeepTC-Enhancer in the context of different maintenance tasks that involve test case navigation. Moreover, from the evaluation with developers, we see that there is a need to improve the suggestions for test case names.

REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson. 2013. Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost. In *Proceedings International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 352–361.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 38–49.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[4] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. 2091–2100.

[5] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[9] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.

[10] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[11] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 179–190.

[12] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. 2015. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 1–38.

[13] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 107–118.

[14] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. 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*. 57–67.

[15] Xavier Devroey, Sebastiano Panichella, and Alessio Gambi. 2020. Java Unit Testing Tool Competition - Eighth Round. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20)*. Seoul, Republic of Korea. https://doi.org/10.1145/3387940.3392265

[16] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.

[17] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42.

[18] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering* 20, 3 (2015), 611–639.

[19] Danielle Gonzalez, Suzanne Prentice, and Mehdi Mirakhorli. 2018. A fine-grained approach for automated conversion of JUnit assertions to English. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*. 14–17.

[20] Robert J. Grissom and John J. Kim. 2005. *Effect sizes for research: A broad practical approach* (2nd edition ed.). Lawrence Earlbaum Associates.

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[22] Manabu Kamimura and Gail C Murphy. 2013. Towards generating human-oriented summaries of unit test cases. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 215–218.

[23] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. *arXiv preprint arXiv:2003.07914* (2020).

[24] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and efficient API misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 192–203.

[25] Fitsum Kifetew, Xavier Devroey, and Urko Rueda. 2019. Java unit testing tool competition-seventh round. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 15–20.

[26] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* (2018).

[27] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[28] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas A Kraft. 2016. Automatically documenting unit test cases. In *2016 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 341–352.

[29] Bin Lin, Csaba Nagy, Gabriele Bavota, Andrian Marcus, and Michele Lanza. [n.d.]. On the Quality of Identifiers in Test Code. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 204–215.

[30] P. W. McBurney and C. McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.

[31] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[32] Urko Rueda Molina, Fitsum Kifetew, and Annibale Panichella. 2018. Java unit testing tool competition-sixth round. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 22–29.

[33] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[34] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[35] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[36] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[37] Annibale Panichella and Urko Rueda Molina. 2017. Java unit testing tool competition-fifth round. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 32–38.

[38] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. 2016. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*. 547–558.

[39] Sarah Rastkar, Gail C Murphy, and Alexander WJ Bradley. 2011. Generating natural language summaries for crosscutting source code concerns. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 103–112.

[40] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.

[41] Giriprasad Sridhara. 2012. *Automatic Generation of Descriptive Summary Comments for Methods in Object-oriented Programs*. University of Delaware.

[42] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 43–52.

[43] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 683–693.

[44] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[45] H. Yonai, Y. Hayase, and H. Kitagawa. 2019. Mercem: Method Name Recommendation Based on Call Graph Embedding. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. 134–141.

[46] Benwen Zhang, Emily Hill, and James Clause. 2016. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 625–636.