Received August 20, 2020, accepted August 30, 2020, date of publication September 9, 2020, date of current version September 24, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3022876

Test Case Understandability Model

NOVI SETIANI[1,2], (Member, IEEE), RIDI FERDIANA[1], (Member, IEEE), AND RUDY HARTANTO[1]

[1]Department of Electrical and Information Engineering, Faculty of Engineering, Universitas Gadjah Mada, Yogyakarta 55281, Indonesia
[2]Department of Informatics, Universitas Islam Indonesia, Yogyakarta 55584, Indonesia

Corresponding author: Novi Setiani (novi.setiani@mail.ugm.ac.id and novi.setiani@uii.ac.id)

ABSTRACT Several automated test case generation techniques have been proposed to date, although the adoption of such techniques in the industry remains low. A key factor that has contributed to this low adoption rate is the difficulty experienced by the developer in terms of reading and understanding automatically generated test cases. For this reason, it is essential to construct a test case understandability model for improving the generated test case. In the present paper, we extracted 20 test case metrics, six developer related metrics and two understandability proxies from a white-box test case classification experiment. Based on these metrics, we employed classification and regression algorithms to build test case understandability model. From the experiment, we can conclude that combined metrics always exhibit better discriminatory performance in classification models as well as a higher correlation in regression models when compared to a model that involved only test case metrics or developer metrics.

INDEX TERMS Test case, understandability model, automated test case generation.

I. INTRODUCTION

Software is increasingly important in all facets of life. Consequently, it's also important to improve software quality by applying software testing practices. There are three activities in software testing [1]: i) test case generation; ii) test case execution; iii) test result evaluation. Exhausted testing will take a lot of time and resources, especially in generating test cases.

Several automated test case generation techniques have been proposed to facilitate the software testing process, including random testing [2], search-based generation testing [3], and dynamic symbolic execution [4]. These techniques only require the source code in order to generate an input test for the program being tested. The output result of the test case execution is evaluated automatically using an assertion statement.

The generation of useful test cases is both a monotonous and an error-prone task. During the development period, the test cases need to be read and understood by different team members. Every test case that generates failure in the program under test (PUT) needs to be investigated so that it can be determined whether the error lies in the PUT or the test case. This process requires the developer to understand the test cases and their behavior.

The associate editor coordinating the review of this manuscript and approving it for publication was Resul Das.

The adoption of automated test case generation techniques in the industry remains low [5]. One reason for this low adoption rate is the fact that automatically generated test cases are more difficult for developers to understand and manage [6], [7] than manually written test cases [8]. After test case generation and execution, developers need to correct every failure that is found based on the execution of the test case. This stage is a manual activity that requires the developer's understanding of the behavior of the test case. In a study conducted by [9], it was reported that developers spend almost 50% of their time trying to understand and analyze the outputs of automated testing tools. It has also been found that the misclassification rate is as high as 20% when the developer has to determine whether the results of the test case execution produced successful or failed outputs [10].

Based on research on [7], [8], [11] it was found that automatically generated tests can be more challenging to understand than manually written tests. For this reason, it is crucial to examine the factors that influence the understandability of automatically generated test cases. To overcome this problem, Daka, *et.al.* [6] proposes an approach in optimizing the presentation of the test cases by developing the readability model. This model can estimate the readability value of a test case. It has been implemented as a secondary fitness function in a search-based test case generation technique to optimize the readability of the resulted test cases.

Daka's evaluation in the developer's understanding found that in 5 out of 10 optimized test cases, there was no change

```
ChainBase chainBase() = new ChainBase();
Command[] commandArray0=
chainBase0.getCommands();
ChainBase1 = new ChainBase(commandArray0);
assertNotSame(chainBase1, chainBase0);
```

FIGURE 1. Example of optimized test case.

in the accuracy of the developer's answers compared to test cases that had not been optimized. They conclude that readability in some test cases correlates with understandability. But, in other test cases, there are factors instead of readability that affect the accuracy of the developer's answers.

Readability optimization generated test cases with a shorter number of LOC compared to the previous test case but with the same behavior. For example, a test case for a ChainBase class consisting of 4 LOCs in the Fig. 1 is an optimization result of a test case with 13 LOCs. However, after being evaluated, the level of accuracy of the developer's response to the optimized test case tends to decrease because the developer is unfamiliar with the assertNotSame statement. Instead of readability, the developer's background in unit testing is also an essential factor that must be considered in improving the automated test case generation approach.

On the developers' side, their performance when classifying white-box test cases has been investigated in terms of whether the output is true (pass) or false (fail) [10]. The experiment involved 106 developers who were asked to classify the outputs of the test cases generated using several methods. The results showed that the majority of developers misclassified the test cases, both those that behaved in the expected fashion and those that exhibited faulty behavior (with a median misclassification rate of 20%).

During the unit testing process, after an error is found in the PUT that is marked by faulty behavior, the developer must read and understand the test cases that have been raised. Thus, it is essential to have a model that can be used to estimate the effort required to understand a given piece of test code. Such a model could be implemented concerning automated test case generation techniques to assess whether or not a generated test case can be understood.

In the present paper, we propose a new understandability model that is specifically designed for automatically generated test cases. The proposed model not only considers the metrics derived from the test cases but also takes into account the metrics derived from the developers. The generated test cases will be adaptive, based on both types of metrics. We extracted 26 metrics and two understandability proxies from a white-box test case classification experiment [11]. We inspected the metrics to examine the degree to which they were correlated with test code understandability. The metrics can be categorized into two types: (i) test-code-readability metrics, and (ii) developer-related metrics. The test-code-readability metrics will be extracted based on the feature selection results derived from the test case readability model [13], while the developer-related metrics will be constructed based on each respondent's background and experience in software testing. There are two models of relevance here, namely the actual binary understandability proxy model and the time actual understandability proxy model. The classification model is built using some classification algorithms: C.45 tree, Bayesian, ANN, and SVM algorithms. Additionally, the prediction model is constructed based on the linear regression, random forest, and SVM regression algorithms.

The remainder of this paper is structured as follows. Section II presents a review of prior work concerning the use of understandability models in software engineering, especially in software testing. Section III outlines our approach to building an understandability model for test case generation. Section IV presents the results of the experiment. Section V discusses the results of the understandability modeling. Finally, section VI draws conclusions based on the results of our experiment and identifies directions for future work.

II. RELATED WORK

This research deals with some topics, namely the test case generation, readability and understandability model in the context of test case.

A. TEST CASE GENERATION

Software testing consists of four main activities—generation of test inputs (test cases), determination of expected outputs, execution of test cases, and verification of test outputs [1]. The execution of test cases is the easiest process to be automated, and there are already frameworks that support this, such as Junit, XUnit, and NUnit [12], [13]. The unit-testing framework can generate a test class skeleton for each class written by the developer and execute it automatically when the program is compiled. However, developers must still complete the contents of the test class by writing the object initiation and calling methods, as well as specifying inputs to execute under test (PUT) programs. The greater the size of the program, the greater the resources needed to write the class content. Several methods have been proposed of generating test cases at the unit level automatically, both via black-box testing, using the random testing method [2], [14], and white-box testing, involving dynamic symbolic execution (DSE) [4], [15] and search-based software testing (SBST) [16]:

1) RANDOM TESTING

Random testing is one of the easiest testing techniques to implement, and it is popular. This method generates random test case inputs, and it is independent of program specifications. However, while random tests are easy to understand and simple to implement, they tend to produce illegal inputs (inputs that conflict with the input limits) and have a low chance of finding a specific input value. Therefore, Pacheco *et al.* [2] proposed random testing directed by feedback from the previous execution results. This method can provide a better code coverage value than conventional random testing does. However, considering the complexity of the software, this method has a low probability of producing quality test data.

2) SYMBOLIC EXECUTION

The symbolic execution method is a test case generation method that executes a program based on symbolic values. Symbolic executions were introduced in the mid-70s, by King [17], Boyer [18], and Howden [19]. Although it can increase the value of code coverage [25] and localization of errors [20], this method still has limitations in the form of an exponential increase in the execution path, as in the case of loops and the completion of complex objects, such as arrays. Therefore, a dynamic symbolic execution is proposed that combines program execution using concrete values and symbolic execution for the specified path [4].

3) MUTATION TESTING

The mutation testing concept was reported by DeMillo *et al.* [21] and Hamlet [22] in late 1978. Mutation testing is an error-based testing technique using testing criteria called mutation score adequacy [23]. This criterion is used to measure the effectiveness of the test case set in detecting errors. Syntactic modifications are made to the original program to create a wrong set of programs (mutants). To assess the quality of the given test cases, these mutants are executed against the test cases, and whether the resulting output is different from the original program is observed. If different outcomes are found, it is concluded that the test case can detect errors in the mutant. The biggest obstacle to adapting mutation testing in unit testing practices is the large computational resources needed to generate and execute quality mutants. Therefore, predictive mutation testing techniques have been developed that can predict the results of mutation testing without executing mutants [24].

4) SEARCH-BASED TESTING

Search-based testing was first published by Miller and Spooner [25] in 1976, which implemented numeric maximization techniques and produced real number values to complete an execution path. In 1990, Korel developed a search algorithm to search for test case inputs by measuring the distance between the target branch and the input execution [26]. Search based software testing is the process of generating test cases using a search-based algorithm based on a particular fitness function. The fitness function is part of the algorithm that guides the search to find the optimal solution. Search-based testing has been compared experimentally with other techniques. The results obtained that this technique has several advantages, namely being able to achieve high levels of coverage and being able to represent various types of inputs (for example, input in the form of vectors) as individuals who are candidates for the solution.

B. READABILITY MODEL

Developers read more program code than writing program code itself [27]. To modify, improve, or add a feature in the software, the developer must read and understand the program code. Therefore, the evaluation of whether the program code is readable or not is critical, especially in the software evolution phase. At present, there are four readability models [28]–[32] have been proposed for predicting whether the program code is readable or unreadable. Code readability is defined as a quality of the code to measure how easy the code is being read and interpreted by the developers.

Buse and Wimer's code readability model [28] is a binary logistic regression classifier, which was built from 120 human evaluation in 100 snippet code. Totally, they have 12000 rows of dataset constructed from 25 structural features of the code and binary assessment about the readability of the code. This model reached more than 80% accuracy in classifying snippet code as ''readable'' or ''not readable''. There are three features with high strength in distinguishing code that is readable and not readable: average number of identifiers, average line length, and average number of parantheses.

Posnett, *et. al.* [29] argued that Buse's may not appropriate with code reading activity in a certain context and they rate on limited size of code. So, they proposed a simpler model to improve upon Buse, et. al model by employing Halstead's metrics and entropy calculation in Buse dataset. They found that code readability is directly proportional to the value of its entropy. So, it's possible to determine the code readability by using only three features: line of code, entropy measurement, and Halstead's metrics. It's supported by the better accuracy of Posnett model than Buse's.

Different from the previous model that relied on syntactic features on code, Dorn [30] proposed another approach based on the assumption that code is read by humans on the screen. Hence, aspects of standard identifier naming, indentation, and syntax highlighting were considered as influencing factor in code readability. Dorn's proposed three new group of metrics: visual, spatial, and linguistic features as addition to structural features. These metrics succeed reach 2.3 times better agreement than previous metrics.

Scalabrino [32] argued that source code lexicon also has impact on code readability beside structural metrics. They proposed a set of textual metrics that measure the consistency between source code and comments, the specificity and the completeness of identifiers. Their finding presents that the combination of structural and textual metrics resulted higher accuracy of code readability models than single metrics. This finding is validated by replicating Buse and Weimer's study and it's confirmed that an increase in readability prediction capability correlates with an increase in the accuracy of FindBugs warning [33].

These previous models focus on predicting readability for source code. In the other side, automatically generation test case is also need to be estimated its readability because they are more difficult for developers to understand

and manage [6], [7] than manually written test cases [8]. Although the test case is written in the form of code, it has specific criteria that make it different from the general source code, such as it contains assertion statement.

Recently, Daka *et al.* [6] researched building readability model specific for the test case. In the study of Daka *et al.* [6], optimization of test cases in terms of readability is evaluated. The objects of this research are 100 classes under test in Java program. Test cases are generated automatically based on a heuristic approach, and test code metrics are computed. Developers are asked to assess test case readability by using a Likert value 1-5. By using feature selection techniques, 20 test code metrics are selected. Then, the readability model is built based on these selected features to estimate test case readability value. The readability model is utilized to optimize previous test cases through minimization techniques.

C. UNDERSTANDABILITY MODEL

Although readable code may relate to its understanding by developer, code readability metrics are not enough to predict whether developers can understand program output, code entities relationships, code semantic and structure. The extent to which the program code can be understood by the developer is influenced by many things because understanding the program is a developer's mental process that requires a high level of code abstraction [137, 24]. Boehm defined software understandability as the quality of software systems for being ease to understood. At present, there have been some metrics proposed to estimate software understandability at the source code level.

The concept of how humans understand entities and products in the field of informatics is explained in cognitive informatics. From this concept, there is derived a cognitive weight of software as a measurement to estimate human efforts in understanding software products based on input, output and internal architecture. Misra and Akman [34] found that only CWCM (Cognitive Weight Complexity Measure) which is modeled by basic control structure is able to meet Weyuker's properties [35] compared with other similar metrics. Besides being simple and language independent, CWCM also provides information about the quality of program design. High complexity indicates that program code is difficult to understand and maintain.

Lin *et al.* [36] proposed a unified understandability model based on Halstad complexity, data spatial complexity, cognitive functional size, number of components, comments ratio and quality of documentation. They used PCA and factor analysis to get the row weight vector, then multiplied the result with the understandability matrix. The complete model understandability model is calculated by using fuzzy integral. Lin *et al.* didn't implement and evaluate the proposed model on an empirical case study.

Different from two previous understandability models that focus on independent language and platform, Thongmak [37] proposed seven objective metrics to estimate understandability value that specific for aspect-oriented software. These metrics are derived from three levels dependency graph mapping of aspect-oriented program code: module-level, class/aspect-level, and system-level. The understandability value resulted as summarization all dependency type that multiplied with expert-determined weights. The threshold of each metrics as a guideline to understandability assessment has not yet been explored.

To conduct research on understandability empirically, some researchers involved programmer or students as their respondents. Shima [38] asked five engineering students (one graduate and four undergraduates) to reconstruct the system and the understandability is evaluated based on their performance. Kasto [39] analyzed the factors that influenced the level of difficulty of code comprehension and code tracing in Java programming exam. The participants were 93 first year students. It was found that cyclomatic complexity, nested block depth and the two dynamic metrics, are significantly correlated to the student performance in code tracing.

Recently, [31] presented an empirical study involving 46 developers and 50 java code. Developers were asked to answer some question related to the code. Correlation analysis was carried out between the understandability and 121 metrics related to source code, documentation and developer profiles. An important finding from this research is effort estimation metrics that has been associated with understandability has a low correlation with real understandability. Therefore, this model cannot be used practically in code understandability improvement.

As shown in Table 1, there are several studies that try to model the understandability software by using readability metrics from code and documentation.

The correlation readability and understandability based on the understandability definition from Boehm as ''a characteristic of software quality which means ease of understanding software systems''. Specific in code understandability as part of a software system, we can define it as a non-trivial mental process that requires building high-level abstractions from code statements or visualizations/models. To understand the source code, the developers need to read it so we can assume that readability is one of the factors affecting code understandability.

TABLE 1. Understandability model.

Author	Metrics	Understandability Proxies	Model
[31]	Code metrics Developer metri cs Documentation metrics	Time and answer accuration	Regression and classification
[39]	Code-related metrics	Answer accuration	Correlation
[38]	Code-related metrics	Reconstruction ability	Probabilistic

D. TEST CASE UNDERSTANDABILITY

Test cases must meet several criteria, such as the size of test case sets [40], and ease of understanding (understandability) [8] to support effectiveness in unit testing. The understandability criterion is important because test cases that are generated automatically have a very long format and are difficult to read [6]. Several studies have proposed improving the understanding of test cases by generating documentation in the form of natural language that contains summaries of test cases [41], simplifying test cases from the size of the LOC[40], and giving the name of the identifier that describes contents of test cases [42].

Understandability measurement is often associated with readability criteria because it is assumed that code is easy to read will be easy to understand. Therefore, a test case readability model has been developed using features extracted from the measurement of several program metrics such as line of code, number of assertions, identifier length, and availability of documentation [6]. The readability assessment was carried out by the survey method to developers who were asked to provide a readability scale for test cases ranging from 1 to 5.

To improve automated test case presentation, Daka optimized the search-based generation test case by using readability evaluation as a secondary fitness function []. Daka has also evaluated whether the readability optimized test case can be better understood by the developer or not [6], [42]. Evaluation of the model was conducted by giving some questions related to the understanding of test cases to 30 students. In terms of time, respondents need a faster time in identifying test cases. However, the level of correctness of respondents' answers to the outcome of the test cases did not change. Therefore, we need a model that describes the test case understandability evaluation as a substitute for the secondary fitness function in the search-based test case generation.

Honfi *et al.* [11] present an exploratory study of how developers classify the resulting white-box tests. The study and its replication were carried out in a laboratory environment by involving graduate students who have understanding and experience in unit testing and programming. They act as junior developers who are testing several classes on unknown large projects with the help of the test case generator tool. They asked to classify The developer's performance is assessed by the accuracy of the test case classification by the developer and the time needed by the developer to classify the test case. These two things are the basis for deriving two understandability proxies to assess the developer's understanding of test cases, namely Actual Binary Understandability and Timed Actual Understandability.

Honfi's research results provide facts that participants tend to misclassify tests, both encoding expected and unexpected behaviors, even if they don't find the task difficult. Developers need time to understand the PUT specifications, test cases, and their execution. Also, it turns out that the classification may require quite a long time, which can slow down the software testing process. Therefore, it is crucial to improve the technique for generating test cases so that they are more easily understood by developers. The first step that must be done is to determine a way to predict whether a test case is easy to understand or not.

Previously, it has been proposed an understandability model for program code, but we need a specific model for the code in the form of generated test cases. This encourages us to conduct the study for building understandability model to decide whether the test case is understandable or not. This research will answer the questions below:

1) RQ1: WHAT IS UNDERSTANDABILITY PROXIES AND METRICS THAT APPROPRIATE FOR CONSTRUCTING A MODEL IN GENERATED TEST CASE?

By considering the collection of proxies measured in terms of code understandability, the purpose of this research question is to give understanding to the research community about the appropriate proxies and metrics that able to assess test case understandability.

2) RQ 2: IS IT POSSIBLE TO DEFINE UNDERSTANDABILITY MODELS ABLE TO PREDICT OR CLASSIFY TEST CASE UNDERSTANDABILITY?

Given a specific test case for a method under test, we want to determine whether the metrics in a model can effectively capture the level of test case understandability.

III. METHODOLOGY

This research was conducted in five steps. First, we extracted the dataset to get test case and developer metrics. We derived the test case understandability proxies from the developers' answers and required time. Then, we prepare the dataset in the form of a matrix as input for classification and regression algorithms. Last, we conduct a model evaluation using some measurements.

A. FEATURE EXTRACTION

The first step in this research is the extraction of Honfi's dataset. We use a subset of the dataset as the main result of Honfi's experiment. This dataset is produced from the experiment of classifying white box generated test cases involving 30 developers and 15 test cases. There are two collections in this dataset, the set of test case files and the results of the experimental evaluation of test cases from the developer. Honfi generates test cases for each class under test by using the Intellitest tool. Evaluation of test cases to the developer is measured by the time needed to understand the test cases and the results of the developer's answers.

The first step in our methodology is extracting these generated test cases to obtain 20 test case metrics, as shown in Table 2. For example, we calculate the method name metric from test case in Fig 2 by counting the number of characters in the test method name: CalculateSumTest284, which is 19 characters. Line of code metrics are measured by counting the number of rows in the test method, which is

```
public void CalculateSumTest284(){
int[] ints = new int[5] {4, 5, 6, 7, 8};
int i = this.CalculateSumTest(0, 0, ints);
Assert.AreEqual<int>(0, i);
}
```

FIGURE 2. Test case example.

TABLE 2. Test case metrics.

Metrics	Description	Data type
Method name	Compute the character length of the method name in the test case	Integer
Line of code	Count the number of rows in the test case	Integer
Constructor	Count the number of constructors calling in the test case	Integer
Exception	The presence of exceptions in test cases	Binary
Identifier ratio	calculate the ratio of the number of identifiers to the number of the string in the test case	Float
Unique identifier	Count the number of unique identifiers in the test case	Integer
Identifiers	Count the number all of the identifier in the test case	Integer
String length	Count the average length of string in the test case	Integer
Assertions number	Count the number of assertions in the test case	Integer
Unique Method	Count the number of unique methods in the test case	Integer
Nulls	The presence of nulls in the test case	Binary
Token entropy	Count the number of token entropy in the test case	Float
Float	The presence of float data type in the test case	Binary
Arithmetic operation	The presence of operation arithmetic in the test case	Binary
Branches	Count the number of branches in the test case	Integer
Numbers	Count the number of number data type in the test case	Integer
Assertions	The presence of assertions in the test case	Binary
Loop	Count the number of the loop in the test case	Integer
method ratio	calculate the ratio of the number of method to the number of the string in the test case	Float
Characters	Count the number of characters in the test case	Integer

five lines. The constructor metric is measured by counting the number of constructors calling that is indicated by the "new" call, which is one calling (new int).

Then, we derived developer-related metrics from the collection of developer profile data as described in Table 3. The first until fifth metrics are collected by using the survey method. The last developer metric, quiz metric, represents the score of quiz containing several questions about unit testing and programming.

TABLE 3. Developer related metrics.

Metrics	Description	Data type
Work	Working experience in the industry in years	Integer
UT	Experience in writing unit testing in years	Integer
CS	Experience in computer science in years	Binary
Prog	Experience in programming in years	Integer
Lang	Experience in C# language in years	Float
Quiz result	Result of a questionnaire about C#	Integer

B. DATA PREPARATION

After 26 metrics from test case files and developer profiles are extracted, the next step is merging these two collections into a matrix $m\ x\ n$. Number of rows, m, shows the count of the test case classification result made by the developer. Thirty developers assess 15 test cases, then $m = 450$ rows. Whereas n is the number of metrics calculated from test cases and developers, where the total is 26.

C. TEST CASE UNDERSTANDABILITY PROXY

In the previous model, the developer understandability is measured by using actual and perceived understandability presented in six understandability proxies [31]. Six proxies are derived in the following context: the developer is given a piece of code, they asked to read the code and answer the question: whether or not understanding the code. Last, they are given some questions related to the code.

In the context of test case understandability, we use two proxies which are derived from two aspects of test code understanding: the *correctness* of the developer's response in understanding evaluation of the test case, and the *time* needed to understand the test code. There are two proxies that able to be extracted from Honfi's *et al.* dataset:

- *Actual Binary Understandability (ABU).* ABU is a binary type variable that is true if the developer can classify the test case output correctly, and false otherwise.
- *Time Actual Understandability (TAU).* TAU is a continuous variable based on the measurement of time spent to classify a given test case (on seconds) by following (1)

$$TAU = \frac{1}{classification\_time} \tag{1}$$

D. CLASSIFIER AND REGRESSION

To build a model for predicting ABU, and TAU we use various classifiers and regression options that are defined in the literature to compare their performance. Specifically, we use the classifier algorithm for modeling ABU with binary class: (i)Decision Tree C.45 (j48), (ii) Bayes Networks, (iii) Supporting Vector Machines (SMO algorithm), and (iv) Multilayer Perceptron Networks. We use a regression algorithm for modeling TAU and NTAU with numeric class: (i)Random Forest, (ii) Linear Regression (iii) Supporting Vector Machines (SMO algorithm), and (iv) Multilayer Perceptron Networks.

ABU data were unbalanced, 75% of data presented the correct answer from the respondent. We use the SMOTE filter [43] on the training sets to get a balanced model by generating artificial instances represent the incorrect answer data. It is crucial to have a compact subset feature, so we used principal component analysis to construct a set of values of linearly uncorrelated features. ABU model is evaluated by using AUC and F-measure for combined metrics, developer-related metrics, and test code metrics. TAU model is evaluated by using Mean Absolute Error and correlation value because the predicted attribute is a numeric type.

To train and test classification and regression models, 10-cross validation techniques are used. It aims to avoid overfitting, the model is only suitable for specifics and cannot be generalized. Data is divided into ten partitions. iteratively, nine partitions are used to build the model, and one partition is used to evaluate the model. Therefore, for each model that is built, it will be tested using data that is always different from the training data.

E. EVALUATION METRICS

For measuring the performance of the classification model, we used AUC and F-measure values. AUC value represents the area under the ROC curve. ROC curves described the performance of models in classifying at different threshold settings. Model performance is represented by the true positive rate (sensitivity) values plotted on the y-axis, and false-positive rate values plotted on the x-axis. AUC values are computed by using integral operation on RUC curves whose values range in 0-1. The higher the AUC value, the better the performance of the model in distinguishing whether a test case can be understood or not. F-Measure is one of the evaluation methods in the classification model that combining recall and precision value. The value of recall and precision in a situation can have different weights. F-measure in (2) presents the reciprocity between Recall and Precision by weighting the harmonic mean of recall and precision.

$$Fmeasure = \frac{2 \times Recall \times Precision}{Recal + Precision} \tag{2}$$

We used the Pearson Correlation and MAE (mean absolute error) in (3) for evaluating the performance of the prediction model. The MAE value represents the average absolute error between the predicted value and the actual value [1].

MAE is mathematically defined as follows:

$$MAE = \frac{1}{n} \sum\nolimits_{i=1}^{n} |f_i - y_i| \tag{3}$$

f_i: predicted value
y_i: actual value
n: number of data.

TABLE 4. Classification using combined metric.

Algorithm	Without PCA		PCA	
	F-measure	AUC	F-measure	AUC
C.45 Tree	0.838	0.859	0.712	0.734
ANN	0.765	0.726	0.690	0.754
Bayesian Net	0.819	0.819	0.628	0.677
SVM	0.666	0.667	0.629	0.633

TABLE 5. Classification using test case metrics.

Algorithm	Without PCA		PCA	
	F-measure	AUC	F-measure	AUC
C.45 Tree	0.771	0.828	0.774	0.802
ANN	0.713	0.777	0.700	0.747
Bayesian Net	0.796	0.850	0.647	0.668
SVM	0.602	0.607	0.581	0.594

Based on formula 1 above, MAE intuitively calculates the average error by giving equal weight to all data ($i = 1, \ldots, n$).

Besides error measurement, we also used the correlation coefficient (R^2) as model performance evaluation. It demonstrates to what extent the model explains the variance of the dataset. So, if the R^2 of a model is 0.50, then approximately half of the observed variation can be explained by the model's inputs.

IV. RESULTS

A. CLASSIFICATION

To evaluate the classification model for ABU proxy, we use combined metrics, test code metrics, and developer-related metrics. Table 4 shows the F-Measure and AUC of the classification of *ABU* when using combined metrics. Surprisingly, all classification models produce better performance when not using PCA. It should be noted that the AUC achieved by J48 and Bayesian Net in classifying test case understandability reached the 'Good' range (0.8-0.9). In other words, J4 and Bayesian Net can distinguish test cases as actually understandable or not understandable. By considering F-measure, it is clear that we can use both models practically for classifying test cases based on its understandability.

However, the results for the test case understandability classification model that uses only test case metrics (Table 5) or developers (Table 6) are generally no better than the combined metrics model. The model built by the metric test code can reach F-measure 0.721 and AUC 0.756 by using the C.45 algorithm. This value is slightly lower when compared to a developer metric model that can reach 0.796 for F-measure and 0.850 for AUC by utilizing the Bayesian Net algorithm. In detail, there is no striking difference in the model's ability to classify test cases into understandable and not understandable test cases. This is indicated by the average TP Rate for the two classes in the range of 0.7-0.8.

TABLE 6. Classification using developer metrics.

Algorithm	Without PCA		PCA	
	F-measure	AUC	F-measure	AUC
C.45 Tree	0.721	0.756	0.716	0.751
ANN	0.687	0.790	0.694	0.732
Bayesian Net	0.699	0.737	0.698	0.745
SVM	0.658	0.659	0.631	0.636

TABLE 7. Regression evaluation.

	Combined metrics		Testcode metrics		Developer metrics	
Algorithm	Correlation	MAE	Correlation	MAE	Correlation	MAE
Random Forest	0.5914	0.0003	0.5828	0.0003	0.128	0.0005
ANN	0.377	0.0005	0.443	0.0004	0.3632	0.0005
Linear regression	0.4014	0.0004	0.3989	0.0004	0.3989	0.0004
SVMReg	0.3578	0.0004	0.3607	0.0004	0.0379	0.0004

B. REGRESSION

To evaluate the regression model for TAU proxy, we use combined metrics, test code metrics, and developer-related metrics. Table 7 shows the Correlation and MAE of the *TAU model* when using combined metrics. It should be noted that the correlation achieved by Random Forest when predicting TAU value is in 'moderate' range (0.5 -0.7). By considering MAE value that only 0.0003, it is possible for us to use this model practically for computing the understandability of the generated test case by using combined metrics.

V. DISCUSSION

In the previous study, Daka [6] showed the readability improvement by constructing the model based on the test code metrics. This readability improvement aims to optimize test case understandability. However, based on the evaluation, it was concluded that the test cases that have been optimized for its readability are not always associated with the increasing of test case understandability. In this research, we conduct the model understandability construction by using two metrics: test code and developer-related metrics. We use test code metrics from Daka [6] and new developer-related metrics. By combining these metrics, we build the regression model to predict the TAU value and classification model for classifying ABU value.

We achieved a higher AUC for the ABU classification model (0.859) when using combined metrics and comparable to the model using developer metrics (0.850) or test code metrics (0.756). When looking at the F-Measure (0.838), it is clear that the ABU model can be used for classifying test cases based on its understandability. This positive result is supported by the fact that the classifier has good results for both positive and negative instances. The precision and recall for the positive class are fairly good (0.827 for precision and 0.846 for recall). When classifying negative instances, this model is also achieving good performance, 0.849 for precision and 0.831 for recall. We present the snippet of the J.48 tree as the ABU classification model in Fig. 3.

TABLE 8. Test case metrics.

No.	Metrics	Value
1.	Identifier	6
2.	Identifier ratio	0.36
3.	Method ratio	0.18
4.	Method name	19
5.	LOC	5
6.	……	……

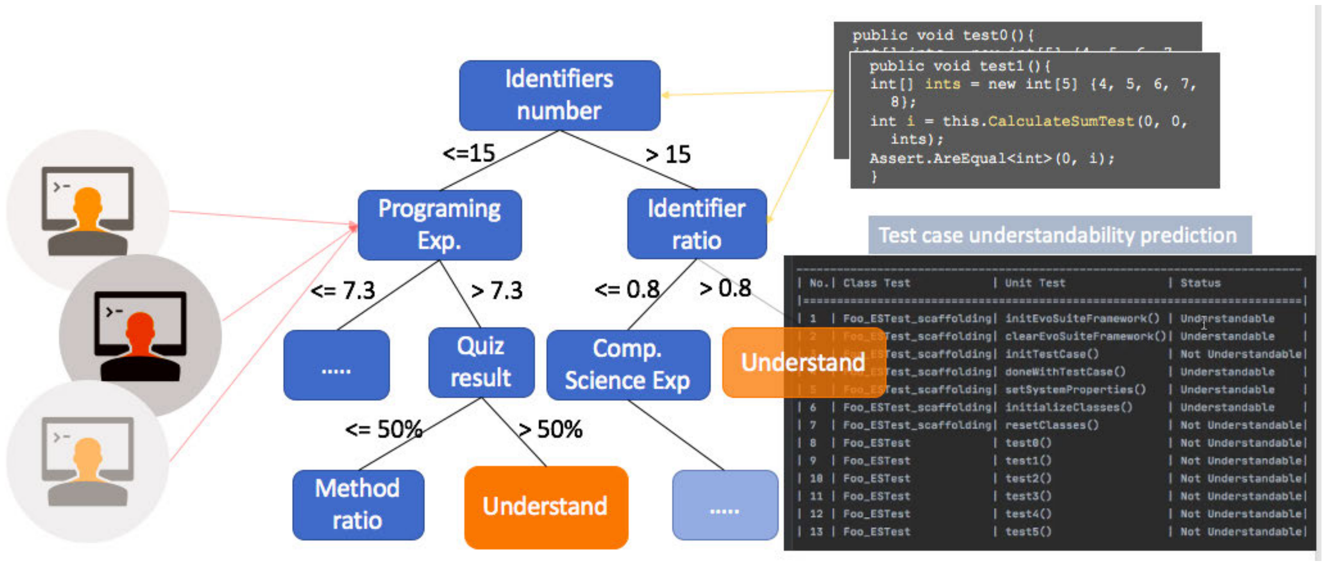
TABLE 9. Developer metrics.

No.	Metrics	Value
1.	Work	5 years
2.	UT	3 years
3.	CS	8 years
4.	Prog	7.5 years
5.	Lang	8 years
6.	Quiz	0.6

The maximum correlation achieved by the TAU regression model when using combined metrics and the Random Forest algorithm. This combination of features achieves a correlation of 0.59 with a root-relative squared error rate of 64.76% and mean absolute error 0.0003. It's comparable to the model constructed by test code metrics that achieve a correlation of 0.582 with a root-relative squared error rate of 64.32%.

Using combined metrics for constructing the understandability model gives the consequence that we must collect data from two different sources: test cases and developers. Of course, it takes more effort than just using a single metric. However, the effort is not too great because it is enough to conduct a questionnaire to the developer once before using this model. To compensate for the computational costs in calculating the test case metric is not too difficult because the file is only traversed once.

To explain examples of using the model, we use the test case in Fig. 2. Decision tree model is used to classify whether the developer with the profile described in will be able to understand the test case or not. The test case feature is extracted to obtain the test case metrics as shown in Table 8 and the developer profile is shown in Table 9. Next, the test case is classified by using the decision tree according to the name of the metric stored in the node and the metric value of the example. The classification process of the test case can be seen in Fig. 4 based on the C.45 model in Fig 3. We can conclude that these test cases will be classified by the developer correctly.



```
public void test0(){
```

```
public void test1(){
int[] ints = new int[5] {4, 5, 6, 7,
  8};
int i = this.CalculateSumTest(0, 0,
  ints);
Assert.AreEqual<int>(0, i);
}
```

```
-------------------------------------------------------------------------------
| No.| Class Test             | Unit Test                 | Status            |
|=============================================================================|
| 1  | Foo_ESTest_scaffolding| initEvoSuiteFramework()   | Understandable    |
| 2  | Foo_ESTest_scaffolding| clearEvoSuiteFramework()| Understandable    |
| 3  | Foo_ESTest_scaffolding| initTestCase()            | Not Understandable|
| 4  | Foo_ESTest_scaffolding| doneWithTestCase()        | Understandable    |
| 5  | Foo_ESTest_scaffolding| setSystemProperties()     | Understandable    |
| 6  | Foo_ESTest_scaffolding| initializeClasses()       | Understandable    |
| 7  | Foo_ESTest_scaffolding| resetClasses()            | Not Understandable|
| 8  | Foo_ESTest             | test0()                   | Not Understandable|
| 9  | Foo_ESTest             | test1()                   | Not Understandable|
| 10 | Foo_ESTest             | test2()                   | Not Understandable|
| 11 | Foo_ESTest             | test3()                   | Not Understandable|
| 12 | Foo_ESTest             | test4()                   | Not Understandable|
| 13 | Foo_ESTest             | test5()                   | Not Understandable|
```

FIGURE 3. **Snippet of C.45 classification model.**

```
if (Identifier) <= 15 then
  if (Prog > 7.3) then
if (Quiz > 0.5) then Right
```

FIGURE 4. **Classification execution.**

Our research is about proposing the test case understandability model by reanalyzing Honif's experiment result. Honfi's research tried to investigate the developer's performance when asked to classify the output of test case execution. This experiment found several factors that influence the developer's success when determining whether the behavior of the testing execution is faulted or expected. Measurement of test case quality in the understandability aspect is needed as one of the prerequisite information in optimizing the test case presentation. We extracted Honfi's data to obtain readability metrics from test cases and developer metrics from the questionnaire results. Expecting that understandability is better captured by a combination of multiple features, we present an analysis of the data from the Honfi study, in which we use different modeling techniques. Further, we construct a binary classifier of understandability based on various interpretable test case features and developer profiles. From this study, there is new knowledge that to estimate whether a developer can understand a test case, we can use a combination of 20 metrics of test cases and six metrics from developers in a random forest-based model. Practically, this model can be used to measure the quality of test cases in the test case generation techniques.

VI. CONCLUSION

We conducted empirical research to analyze test case understandability by exploiting the subset of test case evaluation result (450 instances) from Honfi's study [44] that involved 30 developers and 15 white box generated test cases. We extracted 20 test code metrics from the generated test case and six developer-related metrics from the preliminary survey. We used two understandability proxies, ABU (Actual Binary Understandability) obtained from respondent answers and TAU (Timed Actual Understandability), which is inversely proportional to the time required by respondents to provide answers. To handle the unbalanced data problem (the dataset contained almost 75% positive instances), we utilized SMOTE filtering.

By applying C.45, Bayesian Net, ANN, and SVM algorithm, we build and evaluate classification model to classify the test cases based on its ABU value. We also employed the regression techniques (Random Forest, Linear Regression, ANN, and SVM) to construct a prediction model for the TAU value. Combined metrics always give a better discriminatory performance in the classification model and a higher correlation in the regression model compared to a single metric. Classification model can achieve reliable performance while the regression model gained a moderate performance.

We have presented that our research success in answering the questions that stated previously:

1) RQ1: WHAT IS UNDERSTANDABILITY PROXIES AND METRICS THAT APPROPRIATE FOR CONSTRUCTING A MODEL IN GENERATED TEST CASE?

From the experiment, we can conclude that actual binary understandability is the most appropriate proxy, and combined metrics (readability test case and developer) are the most relevant metrics for constructing the model.

2) RQ 2: IS IT POSSIBLE TO DEFINE UNDERSTANDABILITY MODELS ABLE TO PREDICT OR CLASSIFY TEST CASE UNDERSTANDABILITY?

We can conclude that the model can be used practically in classifying the test case based on the combination of test code metrics and developer-related metrics. In the future, we will employ the classification model as an additional fitness function in white box test case generation to refine the test case understandability.

ACKNOWLEDGMENT

The authors would like to thank D. Honfi and Z. Micskei for sharing their experiment dataset, especially the generated test case.

REFERENCES

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, U.K.: Cambridge Univ. Press, 2016, doi: 10.1017/9781316771273.

[2] C. Pacheco and M. D. Ernst, ''Randoop: Feedback-directed random testing for Java,'' in *Proc. Conf. Object-Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2007, pp. 815–816, doi: 10.1145/1297846.1297902.

[3] G. Fraser and A. Arcuri, ''1600 faults in 100 projects: Automatically finding faults while achieving high coverage with EvoSuite,'' *Empirical Softw. Eng.*, vol. 20, no. 3, pp. 611–639, Jun. 2015, doi: 10.1007/s10664-013-9288-2.

[4] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, ''GRT: An automated test generator using orchestrated program analysis,'' in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 842–847, doi: 10.1109/ASE.2015.102.

[5] A. Arcuri, ''An experience report on applying software testing academic results in industry: We need usable automated test generation,'' *Empirical Softw. Eng.*, vol. 23, no. 4, pp. 1959–1981, Aug. 2018, doi: 10.1007/s10664-017-9570-9.

[6] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, ''Modeling readability to improve unit tests,'' in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2015, pp. 107–118, doi: 10.1145/2786805.2786838.

[7] J. M. Rojas, G. Fraser, and A. Arcuri, ''Automated unit test generation during software development: A controlled experiment and think-aloud observations,'' in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2015, pp. 338–349, doi: 10.1145/2771783.2771801.

[8] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, ''An empirical investigation on the readability of manual and generated test cases,'' in *Proc. 26th Conf. Program Comprehension (ICPC)*, 2018, pp. 348–351, doi: 10.1145/3196321.3196363.

[9] G. Fraser, M. Staats, P. Mcminn, A. Arcuri, and F. Padberg, ''Does automated unit test generation really help software testers? A controlled empirical study,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 1–49, Sep. 2015, doi: 10.1145/2699688.

[10] D. Honfi and Z. Micskei, ''Classifying generated white-box tests: An exploratory study,'' *Softw. Qual. J.*, vol. 27, no. 3, pp. 1339–1380, Sep. 2019, doi: 10.1007/s11219-019-09446-5.

[11] E. Daka and G. Fraser, ''A survey on unit testing practices and problems,'' in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Nov. 2014, pp. 201–211, doi: 10.1109/ISSRE.2014.11.

[12] P. Louridas, ''JUnit: Unit testing and coiling in tandem,'' *IEEE Softw.*, vol. 22, no. 4, pp. 12–15, Jul. 2005, doi: 10.1109/MS.2005.100.

[13] C. Wiederseiner, S. A. Jolly, V. Garousi, and M. M. Eskandar, ''An open-source tool for automated generation of black-box xUnit test code and its industrial evaluation,'' in *Testing—Practice and Research Techniques* (Lecture Notes in Computer Science: Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6303. Berlin, Germany: Springer, 2010, pp. 118–128, doi: 10.1007/978-3-642-15585-7_11.

[14] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, ''An empirical comparison of combinatorial testing, random testing and adaptive random testing,'' *IEEE Trans. Softw. Eng.*, vol. 46, no. 3, pp. 302–320, Mar. 2020, doi: 10.1109/TSE.2018.2852744.

[15] M. Baluda, G. Denaro, and M. Pezze, ''Bidirectional symbolic analysis for effective branch testing,'' *IEEE Trans. Softw. Eng.*, vol. 42, no. 5, pp. 403–426, May 2016, doi: 10.1109/TSE.2015.2490067.

[16] A. Panichella, F. M. Kifetew, and P. Tonella, ''Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets,'' *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, Feb. 2018, doi: 10.1109/TSE.2017.2663435.

[17] J. C. King, ''A new approach to program testing,'' in *Programming Methodology* (Lecture Notes in Computer Science: Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 23. Berlin, Germany: Springer, 1975, pp. 278–290, doi: 10.1007/3-540-07131-8_30.

[18] R. S. Boyer, B. Elspas, and K. N. Levitt, ''SELECT—A formal system for testing and debugging programs by symbolic execution,'' in *Proc. Int. Conf. Reliable Softw.*, 1975, pp. 234–245, doi: 10.1145/800027.808445.

[19] W. E. Howden, ''Symbolic testing and the DISSECT symbolic evaluation system,'' *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 266–278, Jul. 1977, doi: 10.1109/TSE.1977.231144.

[20] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, ''DARWIN: An approach for debugging evolving programs,'' *ACM Trans. Soft. Eng. Method*, no. 19, Jul. 2012, doi: 10.1145/2211616.2211622.

[21] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, ''Hints on test data selection: Help for the practicing programmer,'' *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978, doi: 10.1109/C-M.1978.218136.

[22] R. G. Hamlet, ''Testing programs with the aid of a compiler,'' *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279–290, Jul. 1977, doi: 10.1109/TSE.1977.231145.

[23] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, ''Mutation testing advances: An analysis and survey,'' *Adv. Comput.*, vol. 112, pp. 275–378, Jan. 2019, doi: 10.1016/bs.adcom.2018.03.015.

[24] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, ''Predictive mutation testing,'' *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, pp. 898–918, Sep. 2019, doi: 10.1109/TSE.2018.2809496.

[25] W. Miller and D. L. Spooner, ''Automatic generation of floating-point test data,'' *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 3, pp. 223–226, Sep. 1976.

[26] B. Korel, ''Dynamic method for software test data generation,'' *Softw. Test., Verification Rel.*, vol. 2, no. 4, pp. 203–213, Dec. 1992, doi: 10.1002/stvr.4370020405.

[27] L. Erlikh, ''Leveraging legacy system dollars for e-business,'' *IT Prof.*, vol. 2, no. 3, pp. 17–23, 2000, doi: 10.1109/6294.846201.

[28] R. P. Buse and W. Weimer, ''Learning a metric for code readability,'' *IEEE Trans. Soft. Eng.*, vol. 36 , no. 4 , Jul./Aug. 2010, doi: 10.1109/TSE.2009.70.

[29] D. Posnett, A. Hindle, and P. Devanbu, ''A simpler model of software readability,'' in *Proc. 8th Work. Conf. Mining Softw. Repositories (MSR)*, 2011, pp. 73–82, doi: 10.1145/1985441.1985454.

[30] J. Dorn, ''A general software readability model,'' M.S. thesis, Dept. Comput. Sci., Univ. Virginia, Charlottesville, VA, USA, 2012. [Online]. Available: https://web.eecs.umich.edu/~weimerw/students/dorn-mcs-pres.pdf

[31] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, ''Automatically assessing code understandability,'' *IEEE Trans. Softw. Eng.*, early access, Feb. 25, 2019, doi: 10.1109/TSE.2019.2901468.

[32] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, ''Improving code readability models with textual features,'' in *Proc. IEEE 24th Int. Conf. Program Comprehension (ICPC)*, May 2016, pp. 1–10, doi: 10.1109/ICPC.2016.7503707.

[33] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, ''A comprehensive model for code readability,'' *J. Softw., Evol. Process*, vol. 30, no. 6, p. e1958, Jun. 2018, doi: 10.1002/smr.1958.

[34] S. Misra and I. Akman, ''Comparative study of cognitive complexity measures,'' in *Proc. 23rd Int. Symp. Comput. Inf. Sci.*, Oct. 2008, pp. 1–4, doi: 10.1109/ISCIS.2008.4717939.

[35] E. J. Weyuker, ''Evaluating software complexity measures,'' *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1357–1365, Sep. 1988, doi: 10.1109/32.6178.

[36] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, ''GUITAR: An innovative tool for automated testing of GUI-driven software,'' *Automated Softw. Eng.*, vol. 21, no. 1, pp. 65–105, Mar. 2014, doi: 10.1007/s10515-013-0128-9.

[37] M. Thongmak and P. Muenchaisri, ''Measuring understandability of aspect-oriented code,'' in *Digital Information and Communication Technology and Its Applications* (Communications in Computer and Information Science), vol. 167, no. 2. Berlin, Germany: Springer, 2011, pp. 43–54, doi: 10.1007/978-3-642-2_5.

[38] K. Shima, Y. Takemura, and K. Matsumoto, ''An approach to experimental evaluation of software understandability,'' in *Proc. Int. Symp. Empirical Softw. Eng.*, 2002, pp. 48–55, doi: 10.1109/ISESE.2002.1166925.

[39] N. Kasto and J. Whalley, ''Measuring the difficulty of code comprehension tasks using software metrics,'' in *Proc. 15th Australas. Comput. Educ. Conf.*, Jan. 2013, pp. 59–65, doi: 10.5555/2667199.2667206.

[40] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, ''Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system,'' *Softw. Test. Verification Rel.*, vol. 25, no. 4, pp. 371–396, 2015, doi: 10.1002/stvr.1572.
[41] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, ''The impact of test case summaries on bug fixing performance: An empirical investigation,'' in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 547–558, doi: 10.1145/2884781.2884847.
[42] E. Daka, J. M. Rojas, and G. Fraser, ''Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?'' in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2017, pp. 57–67, doi: 10.1145/3092703.3092727.
[43] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, ''SMOTE: Synthetic minority over-sampling technique,'' *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, Jun. 2002, doi: 10.1613/jair.953.
[44] D. Honfi and Z. Micskei. (2019). *Classifying Generated White-Box Tests: An Exploratory Study*. Accessed: Feb. 11, 2020. [Online]. Available: https://zenodo.org/record/2596044#.XkI4fhMzbUp

NOVI SETIANI (Member, IEEE) was born in East Java, Indonesia, in 1985. She received the B.S. and M.S. degrees in informatics from the Institute Technology of Bandung, Indonesia, in 2012. She is currently pursuing the Ph.D. degree in software engineering with Gadjah Mada University, Yogyakarta, Indonesia.

Since 2012, she has been working as a Lecturer and a Researcher with the Department of Informatics, Universitas Islam Indonesia, Yogyakarta. She actively writes books and articles. Her research interest includes the implementation of data mining in software process. Previously, she researched predicting the software defect by mining the project repository and clustering user requirements. She is also researching a test case generation approach by employing some machine learning algorithms.

RIDI FERDIANA (Member, IEEE) received the Ph.D. degree in software engineering from Gadjah Mada University, Yogyakarta, Indonesia, in 2011.

He has been in the IT industry since 2004 with sufficient experience in the innovation of technology. Since 2008, he has also been working as a Researcher and a Lecturer with the Electrical and Information Engineering, Universitas Gadjah Mada. He gives lectures for many subjects, such as modern software engineering, Web development, project management, interoperability, human–computer interaction, and multimedia technology. He is the author of six books and more than 150 articles. His research interests include technology-enhanced education (e-learning, learning analytics, and MOOC), software engineering (i.e., DevOps, Agile, and software testing), and cloud computing (i.e., cognitive services and Chatbot). In 2007, he awarded as the Microsoft MVP and joined the global innovation program on Microsoft called Microsoft Innovation Center (MIC) Program. On MIC, he holds a responsibility to innovate and to create a better local software economy on Indonesian developers' community. He also received Sandwich Program Grant from ACM Mentor.Net when pursuing his Ph.D. program.

RUDY HARTANTO received the Ph.D. degree in information technology from Gadjah Mada University, Yogyakarta, Indonesia. He is currently an Associate Professor with the Electrical and Information Engineering, Universitas Gadjah Mada. He is an author of more than 100 articles. His research interests include human–computer interaction, computer graphics, and multimedia.