Fault-Prone Module Prediction Approaches Using Identifiers in Source Code

Osamu Mizuno, Graduate School of Science and Technology, Kyoto Institute of Technology, Kyoto, Japan

Naoki Kawashima, Graduate School of Science and Technology, Kyoto Institute of Technology, Kyoto, Japan

Kimiaki Kawamoto, Graduate School of Science and Technology, Kyoto Institute of Technology, Kyoto, Japan

ABSTRACT

Prediction of fault-prone modules is an important area of software engineering. The authors assumed that the occurrence of faults is related to the semantics in the source code modules. Semantics in a software module can be extracted from identifiers in the module. Identifiers such as variable names and function names in source code are thus essential information to understand code. The naming for identifiers affects on code understandability; thus, the authors expect that they affect software quality. In this study, the authors examine the relationship between the length of identifiers and existence of software faults in a software module. Furthermore, the authors analyze the relationship between occurrence of "words" in identifiers and the existence of faults. From the experiments using the data from open source software, the authors modeled the relationship between the fault occurrence and the length of identifiers, and the relationship between the fault occurrence and the word in identifiers by the random forest technique. The result of the experiment showed that the length of identifiers can predict the fault-proneness of the software modules. Also, the result showed that the word occurrence model is as good a measure as traditional CK and LOC metrics models.

Keywords: *Fault-Prone Module Prediction, Random Forest, Software Engineering, Source Code Identifier, Word Occurrence Model*

1. INTRODUCTION

Faults in software are not evenly distributed to the modules but are concentrated to specific modules (Hata, Mizuno, & Kikuno, 2012). Many studies pointed out that the 20% of modules include 80% of faults (Fenton & Neil, 1996, 1999). We call such fault-injected modules as fault-prone modules. This fact indicates that if we can predict fault-prone modules correctly, the efficiency of testing improves more. For example, Khoshogoftaar said that if we can

DOI: 10.4018/ijsi.2015010103

predict fault-prone module, we can reduce cost on software testing by half (Khoshgoftaar & Seliya, 2004).

As for the measures for fault-prone module detection, various metrics have been proposed so far (Hata et al., 2012). For example, CK metrics in Object-oriented code (Briand, Melo, & Wust, 2002; Gyimóthy, Ferenc, & Siket, 2005), process metrics (Ostrand, Weyuker, & Bell, 2005), software structures (Graves, Karr, Marron, & Siy, 2000), the metrics from static analysis (Nagappan & Ball, 2005; Zheng et al., 2006), and other metrics.

Faults are injected in software for various reasons. For predicting faults, it is necessary to make an assumption on the relationship between the cause of faults and the existence of faults. In this study, we made an assumption that the semantics in the software module has an effect to the fault existence. Semantics in a software module can be extracted from identifiers in the module.

Identifiers used in source code, such as names of class, method, function, and variable, can provide important information for readability and understandability with meanings of words in identifiers (Deissenboeck & Pizka, 2006). The length of identifiers, the way of naming, and the way of abbreviations of identifiers thus affect to the quality and understandability of source code (Butler, Wermelinger, Yu, & Sharp, 2009; Lawrie, Morrell, Feild, & Binkley, 2006).

Yamamoto et al. adopted the number of variable names as predictor variables of fault density prediction models (Yamamoto, Kamei, Matsumoto, Monden, & Matsumoto, 2009). Their approach used identifiers but did not use words in identifiers. It can be said that identifiers are one of the most important elements of the source code, and thus useful information can be obtained by analyzing the identifiers.

To investigate, we built a model to determine faulty modules using a machine learning technique from the number of occurrences of the identifiers. Using the fault-prone module detection model, we can investigate the relationship between the length of identifiers and software faults in a software module. For further investigation, we analyze the relationship between occurrence of "words" in identifiers and the existence of faults.

The rest of this paper is organized as follows: In Section 2, we describe an importance of identifier. In Section 3, we state the research questions in this study. In Section 4, we describe the first experiment to investigate the relationship between the length of identifiers and fault-proneness. We then describe the second experiment to investigate the relationship between word occurrence and fault-proneness in Section 5. In Section 6, we discuss the result of experiments and find the answer to the research questions. Finally, we conclude this paper in section 7.

2. IMPORTANCE OF IDENTIFIERS

It is said that nearly 70% of the source code of a software system consists identifiers (Deissenboeck & Pizka, 2006). Hence, we expect that the identifiers are key aspects in the source code analysis. Other research such as (Chen, Thomas, Nagappan, & Hassan, 2012) also focuses on the identifiers of source code.

Identifiers are a string of a name of function or variable except reserved words and literals. Usually, programmers often generate identifiers by a combination of words, such as "getPassword", "doCalclationProcess", and so on. A function name by a combination of words that adequately describe the contents of the function can be understood without reading the contents of the code and annotation (comments). When we look at the function call in the code, it is possible to understand what the function does without looking at the body of the function if the function name is acceptable. In cases of variables, with a variable name by combining the words that describe the contents of the stored value and type information, developers can easily understand the meaning and role of variable. In this way, identifiers have useful information to understand the contents of source code as well as the comment lines.

The understandability and the readability are as a part of software quality, and we assume that they deeply influence the existence of faults in software modules.

Characteristics of identifiers such as the length of the string (the length of identifier), the sequence of letters, uppercase, lowercase and the way of abbreviation have become a difference in the naming of identifiers for each programmer or programming rule. Such difference relates to the quality and the readability of source code (Butler et al., 2009; Lawrie et al., 2006). We therefore assume that a good guideline to name identifiers improve the quality of the source code. The reference (Relf, 2004) has suggested a naming guideline for identifiers in Java and Ada.

Characteristics of identifier's naming are classified as follows:

1. Length of identifier;
2. The number of words;
3. Abbreviation;
4. Hungarian notation;
5. Uppercase and lowercase;
6. Numeric words or numbers.

Length of identifier and the number of words in the identifier have a relationship to readability of identifiers. For example, the number of words becomes larger, the more information can be read and the more readability achieved. On the other hand, the number of words become smaller, the less information obtained and the less readability achieved. However, too long identifiers have less readability since it cannot be shown in 1 line of screen.

It is possible to shorten the length of identifiers by omitting the word abbreviations and initials. If developers are not familiar with programming and software development, the code readability decreases. Since the way of abbreviation differs from developers, it could be affected to the readability of source code. Hungarian notation is a naming convention with a prefix that represents the range of scope and type information of identifier. Although it is possible to obtain type information about the identifier, it may mitigate the readability of the source code if the notation is not consistent. Uppercase and lowercase of identifier are elements for the appearance of identifiers, and they affect the readability of source code, too. Numeric words and numbers are often used to create a new identifier that is similar to existing one by adding numbers to the end of an identifier, such as "author" and "author1". This, however, may cause to take different identifiers by mistake. Thus, this also affects the readability.

In this study, we first focus on the length of identifiers in the above characteristics. It is one of the most basic characteristics of the identifiers. The length of identifier can be obtained regardless of the language and easy to measure from source code.

On the other hand, we assume that the semantic information of identifiers also affects to the fault occurrence in software modules. In order to see such semantic information, we conducted an experiment to predict faulty modules using the words themselves in identifiers.

3. RESEARCH QUESTIONS

In order to clarify the objective of this research, we state the following research questions to be confirmed in this research:

RQ1: Can the length of identifiers contribute to predict the quality of software?
RQ2: Can the words included in identifiers of source codes predict the quality of software?

We assume that the length of identifiers can contribute to predict the quality of software. In this study, we target on fault-prone module prediction approaches. We need to show how the length of identifier and the words itself contribute to the fault-proneness compared with the conventional software metrics.

If these two research questions are true, then the following research questions arise:

RQ3: Are there specific lengths of identifier that contribute to predict the quality of software?

RQ4: Are there specific words that contribute to predict the quality of software?

In this paper, we seek the answers for these four questions through experiments using the OSS software repositories.

4. EXPERIMENT 1: LENGTH OF IDENTIFIERS

4.1. Target Projects

In this experiment, we used the source code from two open source software projects, Eclipse and Netbeans, which were prepared for the MSR challenge 2011. Both Eclipse and Netbeans are the integrated development environment (IDE) software, and they are developed using the Java language. Table 1 shows a summary of two projects. For this study, we used entire data set rather than release set. By using the entire data, we can show the possibility of applying this approach to any point of the development. However, we will try to use release data in future since using the release data is more popular and comparable to other studies.

Both projects have developed for about ten years. Thus, the number of software modules in both projects is larger than 300,000, and approximately 50% of modules is faulty module. In this study, a software module means a java class file. To determine faulty status of modules, we used the SZZ algorithm (Sliwerski, Zimmermann, & Zeller, 2005).

Figure 1 shows the distribution of length of identifiers in Eclipse. Let us explain the case of Eclipse in Figure 1. In Figure 1 (a), the existence of faults in Eclipse divided by the number of occurrences of the identifier in all modules is shown for each length of identifiers. Because of the large difference in the number of occurrences for each length of identifiers, we used logarithmic vertical axis in Figure 1 (a). Figure 1 (b) shows the percentage of the existence of faults for each length of identifiers. Note that the percentage of faulty identifiers is different from the percentage of faulty modules in a project. This is due to the difference from the size of modules. Our investigation shows that the average LOC of faulty and non-faulty modules are 1,534 and 330, respectively. Then, the ratio by LOC of faulty modules is about 82%. Since the number of identifiers and LOC has high correlation, we can see that about 80% of identifiers are in faulty modules.

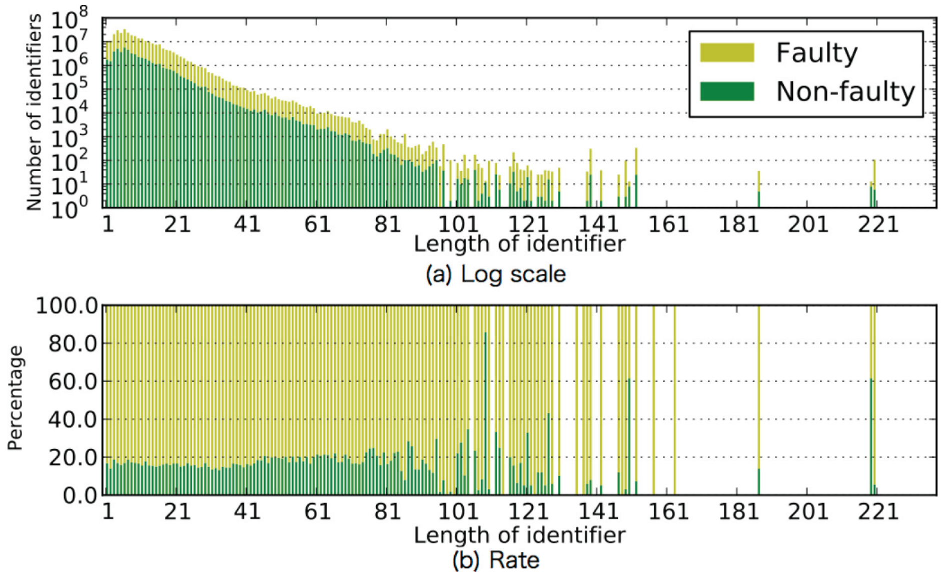
From Figure 1(a), we can see that identifiers with length of 5 to 10 appear most frequently. In the area of length > 10, if the length of identifiers is longer, the number of occurrence of identifiers is smaller. On the other hand, as shown in Figure 1(b) the percentages of faulty modules among the length of identifiers are constant in frequent identifiers. We can observe that the percentages vary in not very frequent identifiers.

Since the above trends are almost the same in a case of NetBeans, we omit graphs

Table 1. Target projects

	Eclipse	Netbeans
# modules	311248	393524
# faulty modules	150289	226621
Average length in non-faulty module	8.98	8.55
Average length in faulty modules	9.15	8.54
The longest length of identifier	237	210

Figure 1. Distribution of identifier length (Eclipse)



in NetBeans. As shown in Table 1, in both faulty and non-faulty modules, average length of identifiers is almost equal. It is thus difficult to predict the faulty modules using the length of identifiers only. In this study, we utilize the distribution of the length of identifiers for each module.

4.2. Definition of Metrics

We defined the following metrics in this study from the viewpoint of the length of identifiers:

- **OC(l):** The number of occurrences of identifiers with length of *l* in a module;
- **TN:** The total number of identifiers found in a module;
- **LOC:** The lines of code of a module. Empty lines are excluded.

The maximum *l* for Eclipse is 237 and for NetBeans is 221.

4.3. Fault-Prone Prediction Model

In order to find the answer to the RQ1, we built a fault-prone module prediction model. Figure 2 shows a flow chart of experiments. We extract both faulty and non-faulty source code modules. For each module, we count up the $OC(l)$ and TN for each module. In this study, we aim at the fault-proneness of the modules. The main reason is a simplicity of the problem. We can extend this study for the number of faults in modules in a future work since the number of faults in a module can be collected in both Eclipse and Netbeans.

For the comparative study, we also measure the length of code (*LOC*). We then build the following two models to predict fault-prone modules using the random forest (Breiman, 2001) technique: (Ex. A) A model using the number of occurrences of identifiers, and (Ex. B) A model using the lines of code. It is widely known that the size of software and the existence of faults are deeply related. We thus use a model with *LOC* for the target of comparison.

4.4. Evaluation Measures

Here, we describe evaluation measures of experiments. The result of experiment is shown in the form of Table 2:

Figure 2. A flow chart of experiment 1

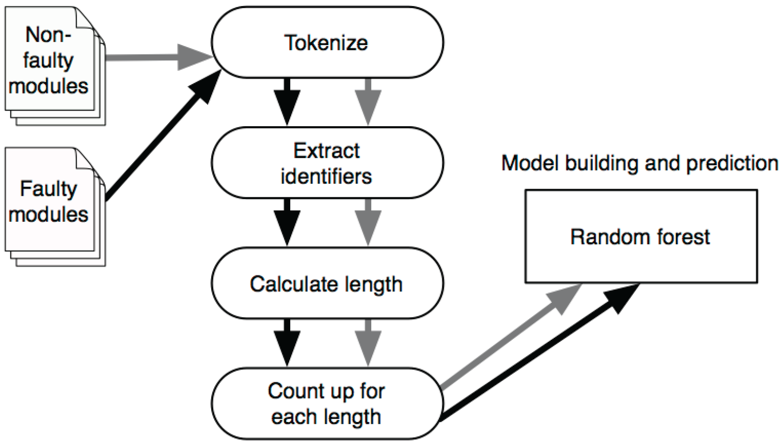


Table 2. Confusion matrix of a result of experiment

		Actual	
		Non-Faulty	**Faulty**
Prediction	Non-faulty	TP	FP
	Faulty	FN	TN

- **True Positive (TP):** True positive shows the number of modules that are classified as fault-prone which are actually faulty;
- **False Positive (FP):** False positive (FP) shows the number of modules that are classified as fault-prone, but are actually non-faulty;
- **True Negative (TN):** True negative (TN) shows the number of modules that are classified as non-fault-prone, and are actually non-faulty;
- **False Negative (FN):** False negative shows the number of modules that are classified as not fault-prone, but are actually faulty.

In order to evaluate the results, we prepare three measures: recall, precision:

- **Recall:** Recall is the ratio of modules correctly classified as fault-prone to the number of entire faulty modules. Recall is defined as follows:

$$Recall = \frac{TN}{TN + FN}$$

- **Precision:** Precision is the ratio of modules correctly classified as fault-prone to the number of entire modules classified fault-prone. Precision is defined as follows:

$$Precision = \frac{TN}{FP + TN}$$

- **F Measure:** Since recall and precision are in the trade-off, F-measure is used to combine recall and precision. F-measure is defined as follows:

$$F = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

4.5. Results

Tables 3 and 4 summarize the results of prediction by two models using evaluation measures such as precision, recall, and F-measure. From Table 3, we can see that predictions are done with about 80% of *F* measure in both Eclipse and Netbeans. We can also see that the precision is relatively low in Netbeans.

Let us compare the results using the length of identifiers and the *LOC*. We can see that all the measures in Table 3 show higher values than that of Table 4.

Here, one question arises. Is the result in Table 3 affected by the size of the source code? As shown in Table 4, the size of the source code itself is a relatively strong predictor.

In order to eliminate the effect of size, we performed an additional experiment (Ex. C). In this additional experiment, we counted the total number of identifiers, TN, in a module. The occurrence of identifiers $OC(l)$ is then normalized by TN. For the prediction of fault-prone modules, we used $OC(l)/TN$ instead of $OC(l)$.

Table 5 shows the result of the additional experiment. By comparing Table 3 and Table 5, we can see that the normalized occurrence of identifier length shows better result than the original one.

Table 6 shows the ranking of mean decrease Gini for each length of identifiers in both Eclipse and Netbeans from the third experiment. The mean decrease Gini is a measure to evaluate the importance of variables used in the random forests. High mean decrease Gini value means that the variable is important. From Table 6, identifiers with shorter length have higher mean decrease Gini values.

Table 3. Ex. A: Result of prediction using occurrences of identifier lengths (average of 10 times)

	Precision	Recall	F
Eclipse	0.866	0.913	0.889
Netbeans	0.765	0.865	0.812

Table 4. Ex. B: Result of prediction using the lines of code (average of 10 times)

	Precision	Recall	F
Eclipse	0.760	0.736	0.748
Netbeans	0.577	0.665	0.618

Table 5. Ex. C: Result of prediction using the normalized occurrence of identifier lengths (average of 10 times)

	Precision	Recall	F
Eclipse	0.873	0.919	0.896
Netbeans	0.773	0.870	0.819

Table 6. Top 10 mean decrease Gini for the model using the identifier length (average of 10 times)

Rank	Eclipse		Netbeans	
	Length	MDG	Length	MDG
1	19	5421	20	5763
2	20	4483	1	5726
3	13	4461	2	5637
4	22	4444	17	5488
5	17	4444	16	4937
6	16	4354	19	4893
7	15	4334	19	4390
8	21	4113	14	4324
9	14	4050	15	4297
10	11	3988	22	4091

5. EXPERIMENT 3: COUNT OF WORDS

5.1. Procedure

The process of proposed method is shown as follows:

1. We extract identifiers from source code, and split identifiers to words using a tool "lscp";
2. We count the number of occurrence of words in a module;
3. We link the existence of fault in a module and the counted number of words in a module;
4. By a machine leaning technique, we make a fault-prone predicting model using the number of words in a module.

Figure 3 shows a chart of the process of making fault-prone module prediction model.

5.2. Target Projects

In this experiment, the projects used in the Experiment 1 are too large to apply the word extraction. We therefore used other open source software for this experiment 2. For collecting target projects, we used GitHub3 and PROMISE Data repository.

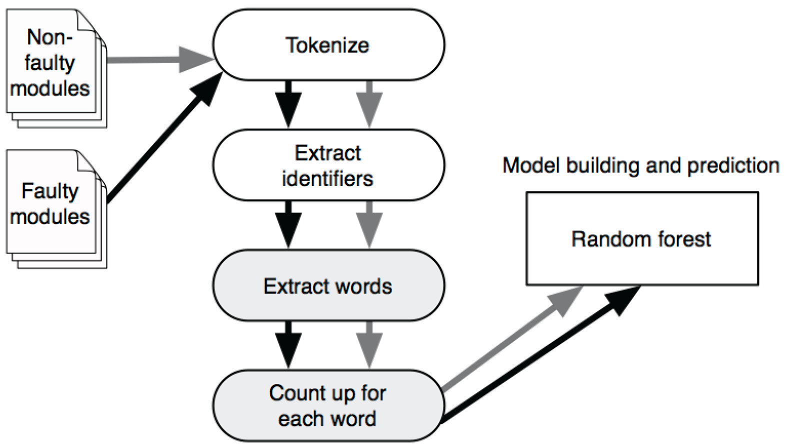
The PROMISE data repository is an open repository of data sets for software engineering. We used data sets in PROMISE repository as target projects, which include defect data, CK metrics (Chidamber & Kemerer, 1994), and LOC. Since PROMISE repository does not store source code repositories, we need to obtain source code repositories of corresponding projects from other repository. We got repositories of target projects from GitHub.

The data set from PROMISE is written defect data with corresponding class names and revisions as well as CK metrics and LOC. We linked defect data from PROMISE to source code from GitHub based on the class names and revisions in the data set. We ignored the files that cannot be linked with defect data during this process.

In this experiment, we used Apache Ant and Apache Xalan projects as targets:

- **Apache Ant:** A software tool for automating software build processes. It is implemented using the Java language;
- **Apache Xalan:** A software tool that implements the XSLT 1.0 XML transformation language and the XPath 1.0 language.

Figure 3. A flow chart of experiment 2



These target projects are written in Java language. In this experiment, we use learning data and testing data in a project to see the performance of prediction. For this purpose, we divided data into learning and testing by revisions. Table 7 shows the number of learning files and test files of target projects, and also shows the versions of learning files and test files in the target projects.

Table 8 shows the number of fault injected files and fault free files in the target projects. These data is obtained from the PROMISE data set. Table 9 shows the number of unique words in each project.

5.3. Splitting an Identifier to Words

Here, we explain a rule to split an identifier to words. There are several ways of connecting words in an identifier. Sometimes they are joined by figures or symbols, in the other case, a word

Table 7. Training data and testing data of target projects

Project	Training		Testing	
	Rev.	# Files	Rev.	# Files
Ant	1.4, 1.5, 1.6	819	1.7	741
Xalan	2.4	682	2.5	762

Table 8. Fault data of target projects

Project	Training		Testing	
	# Faulty	# Non-Faulty	# Faulty	# Non-Faulty
Ant	164	655	166	575
Xalan	110	572	387	375

Table 9. Number of unique words

Project	# Unique Words
Ant	2726
Xalan	4295

is connected to a word without any connection part. The following shows examples:

- **A case of connection using symbols:** For example, "max_size" should be divided into "max" and "size";
- **A case of camel case:** For example, "maxSize" should be divided into "max" and "size";
- **A case of connection by using a figure:** For example, "int2char" should be divided into "int", "2", and "char";
- **A case of connecting without any figures or symbols:** For example, "maxsize" should be divided into "max" and "size".

In this study, we can handle identifiers in cases of above 1 and 2. For splitting the identifiers in the source code, we used the tool "lscp".

5.4. Fault-Prone Prediction Model

As in the experiment 1, we use the random forest technique for the machine-learning algorithm. Since the random forest technique can handle thousands of input variables, this technique is appropriate to analyze the occurrence of words in source code. When analyzing occurrence of words for identifiers, we need to handle a lot of identifiers in source code. In order to apply the random forest technique, we used a statistical software R.

5.5. Comparison with Other Prediction Method

Since we used the data set published in PROMISE, we can obtain CK metrics and LOC than the word occurrence. By using these data, we can perform a comparative experiment. The metrics for comparison are CK metrics and *LOC*, which are both collected in Apache Ant and Xalan projects.

For the object-oriented design, Chidamber and Kemerer proposed an object-oriented metrics suit. The metrics suit is called "CK metrics". CK metrics suit includes the following 6 metrics (Chidamber & Kemerer, 1994):

- ***W MC* (Weighted Methods per Class):** The number of methods defined in each class;
- ***DIT* (Depth of Inheritance Tree):** The number of ancestors of a class;
- ***NOC* (Number of Children):** The number of direct descendants for each class;
- ***CBO* (Coupling between Object classes):** The number of classes to which a given class is coupled;
- ***RFC* (Response for a Class):** The number of methods that can be executed in response to a message being received by an object of that class;
- ***LCOM* (Lack of Cohesion of Methods):** The number of pairs of member functions without shared instance variables, minus the number of pairs of functions with shared instance variables. If this subtraction is negative, the metric is set to zero.

5.6. Results

We describe observations on each evaluation measure. This measure evaluates the success of the prediction.

Tables 10 and 11 show that word occurrence based fault prediction achieves higher recall than LOC in both projects. They also show that the proposed method achieves higher recall but lower precision than CK metrics in Xalan project (See Table 11). On the other hand, in Ant project, the proposed method achieves lower recall but higher precision than CK metrics (See Table 10). The values of *F* measure also vary in projects.

5.7. Importance of Words

Figure 4 and Figure 5 show the Gini index of top 10 words in Ant and Xalan, respectively. These figures show that important word include “java”, “except”, “apach” in both projects. Among them, “except” is the third highest word in Ant and the second highest in Xalan. The word “except” is a stemmed form and it includes both “exception” and “except”. From this fact, we can say that faults frequently appear in the modules that are related to the exception.

6. DISCUSSION

6.1. RQ1 and RQ2: Fault-Prone Prediction by Length of Identifiers and Word Occurrence

First, we intend to find the answer of the research question RQ1 and RQ2. As shown in Table 2, the result of prediction shows that the length of identifiers can be a measure for fault-prone module prediction. We can thus conclude that the answer of RQ1 is “yes”. The result of proposed method in Tables 10 and 11 show that proposed method can predict fault-prone module as well as the CK metrics and LOC. Therefore the count of word included identifiers of source codes predict the quality of software. We can thus conclude that the answer of RQ2 is “yes” and it is confirmed at a certain degree.

6.2. RQ3: Specific Length of Identifiers

Next, we intend to find the answer of the research question RQ3. From Table 6, we can see that the length 20 has high mean decrease Gini in both Eclipse and Netbeans.

Figure 6 shows an example of identifiers with length 20. Identifiers with length 20 appear 3,574,737 times in Eclipse as shown in Figure 1 (a). The unique number of identifiers with length 20 is 9,559. In this length, we observed that there are many method and class names, but not so much variable names.

Let us examine Table 6 in more detail. As for the case of Eclipse, long identifiers that have lengths larger than 11 have high mean decrease Gini. As for the case of Netbeans, long identifiers that have lengths larger than 14 show high mean decrease Gini, too. Very short identifiers with lengths 1 and 2, however, have strong effect to the fault-proneness in Netbeans.

Table 10. Result in Ant project

	Precision	Recall	F
Word occurrence	0.606	0.668	0.636
CK metrics	0.631	0.656	0.643
LOC	0.522	0.650	0.579

Table 11. Result in Xalan project

	Precision	Recall	F
Word occurrence	0.142	0.625	0.232
CK metrics	0.127	0.700	0.214
LOC	0.127	0.551	0.206

Figure 4. Importance of words (Ant)

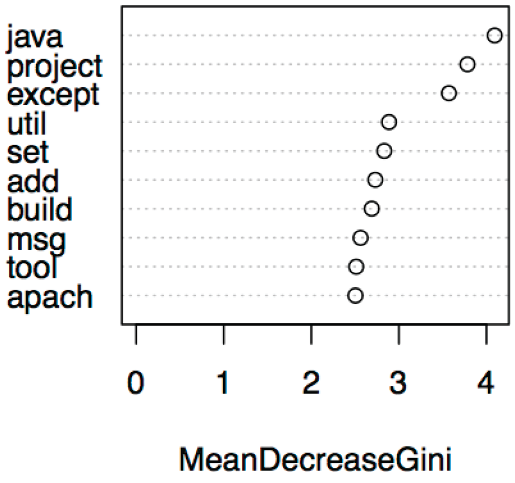


Figure 5. Importance of words (Xalan)

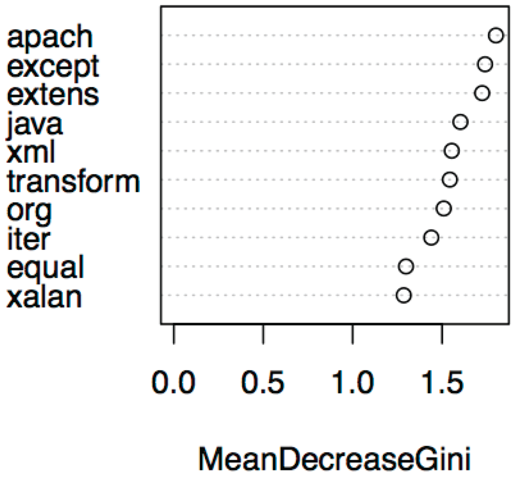


Figure 6. A part of identifiers with length 20 in Eclipse

```
testIdentifierBefore  EVENT_PREFERRED_SIZE
getConstantPoolCount  EmptyCommentDetector
XmCreateScrolledText  getProgressIndicator
calculateChevronTrim  TemplatesPage_remove
addToTopLevelFilters  isConstantTypeEntity
testEnumHighlighting  testAddParenthesis01
SearchPart_learnMore  checkNodeAndChildren
setProxyAuthenticate  DECORATOR_ARRAY_SIZE
IReconcilingStrategy  canRemoveParenthesis
fNavigateActionGroup  P_COMMIT_SET_ENABLED
```

We can conclude that the occurrence of identifiers with length 20 has specific influence to the fault-proneness of the module. Furthermore, this result indicates that identifiers with length from 10 to 25 have stronger impact to the fault-proneness rather than other lengths. From the result of experiments, we can thus conclude that the answer to the RQ3 is “yes”.

6.3. RQ4: Important Words in Identifiers

As shown in Tables 10 and 11, we can also detect important words for predicting fault prone modules at a certain extent. For this reason, we can say that the answer to RQ 4 is partially “yes”. However, several problems remain at this point. For example, we need to determine appropriate stop words for this approach. As shown in Tables 10 and 11, there are words like “java” and “apach(e)” which are commonly appears in most of source code in Apache foundation’s source code. Much precise analysis should be done in a future work.

6.4. Threats to Validity

- **Bugs in Programs:** In the process of the experiment, we built programs of the tokenization of the source code, extracting the identifier, and calculation of the length of identifiers. Any bugs in programs may result the incomplete data and derive not precise results. Fortunately, we have not found such serious bugs so far;
- **Incomplete Training Data:** Faults information in software modules used for modeling the training data are computed by an implementation of the SZZ algorithm (S ´liwerski et al., 2005). In nature, since the SZZ algorithm cannot handle all bugs in a bug tracking system and the software repository, there is a room to improve the algorithm. The data obtained by the SZZ algorithm is, therefore, not complete. However, to our best knowledge, the SZZ algorithm is one of the best solution to integrate the bug tracking system and the repository;
- **Specific Projects:** In this study, we used only two OSS projects, Eclipse and Netbeans. Even though they are large and well-known projects, we cannot generalize the result in this study at this point. We need to have more experiments for further studies to generalize the result in this study.

7. CONCLUSION

In this study, we investigated the relationship between the length of identifiers and software faults in a software module. The results showed that there is a certain relationship between the length of identifier and existence of software faults. We can also conclude that count of words can predict fault-prone modules and proposed method is not less than common method in predicting efficient, but proposed method is less than common method in time of machine learning.

For the future issue, if we reveal that tendency of fault-prone words, we know rule of naming identifier that improve quality of source code. In the other, if we choice word which used in independent variable, we can improve the predicting efficient and speed up machine leaning.

REFERENCES

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32. doi:10.1023/A:1010933404324

Briand, L. C., Melo, W. L., & Wust, J. (2002). Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, *28*(7), 706–720. doi:10.1109/TSE.2002.1019484

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2009). Relating identifier naming flaws and code quality: an empirical study. In *Proc. of the Working Conf. on Reverse Engineering* (p. 31-35). IEEE Computer Society. doi:10.1109/WCRE.2009.50

Chen, T.-H., Thomas, S. W., Nagappan, M., & Hassan, A. E. (2012). Explaining software defects using topic models. In *Proc. of 9th Working Conference on Mining Software Repositories (MSR2012)* (pp. 189-198). Academic Press.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. doi:10.1109/32.295895

Deissenboeck, F., & Pizka, M. (2006, September). Concise and consistent naming. *Software Quality Control*, *14*(3), 261–282. doi:10.1007/s11219-006-9219-1

Fenton, N. E., & Neil, M. (1996). Predicting software quality using bayesian belif networks. In *Proc. 21st Annual Software Engineering Workshop* (pp. 217-230). Academic Press.

Fenton, N. E., & Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, *25*(5), 675–689. doi:10.1109/32.815326

Graves, T. L., Karr, A. F., Marron, J., & Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Trans. on Software Engineering, 26*(7), 653-661. doi: http://doi.ieeecomputersociety.org/ 10.1109/32.859533

Gyimóthy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, *31*(10), 897–910. doi:10.1109/TSE.2005.112

Hata, H., Mizuno, O., & Kikuno, T. (2012). A systematic review of software fault prediction studies and related techniques in the context of repository mining. *JSSST Computer Software*, *29*(1), 106–117.

Khoshgoftaar, T. M., & Seliya, N. (2004). Comparative assessment of software quality classification techniques: An empirical study. *Empirical Software Engineering*, *9*(3), 229–257. doi:10.1023/B:EMSE.0000027781.18360.9b

Lawrie, D., Morrell, C., Feild, H., & Binkley, D. (2006). What's in a name? A study of identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension* (pp. 3-12). Washington, DC: IEEE Computer Society. doi:10.1109/ICPC.2006.51

Nagappan, N., & Ball, T. (2005). Static analysis tools as early indicators of pre-release defect density. In *Proc. of 27th International Conference on Software Engineering* (pp. 580–586). New York, NY: ACM. doi:10.1145/1062455.1062558

Ostrand, T., Weyuker, E., & Bell, R. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, *31*(4), 340–355. doi:10.1109/TSE.2005.49

Relf, P. A. (2004). *Achieving software quality through identifier names*. Qualcon.

Ś liwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes? (on Fridays.). In *Proc. of 2nd International Workshop on Mining Software Repositories* (pp. 24–28). Academic Press.

Yamamoto, H., Kamei, Y., Matsumoto, S., Monden, A., & Matsumoto, K. (2009). An analysis of relationship between software bug and variable name. Technical Report of IEICE, 109, 67-71.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P., & Vouk, M. A. (2006). On the value of static analysis for fault detection in software. *IEEE Trans. on Software Engineering, 32*(4), 240–253. doi: 10.1109/TSE.2006.38