Do Comments Explain Codes Adequately?

Investigation by Text Filtering

Yukinao Hirata
Kyoto Institute of Technology
Matsugasaki Goshokaido-cho, Sakyo-ku
Kyoto 606-8585, Japan
y-hirata@se.is.kit.ac.jp

Osamu Mizuno
Kyoto Institute of Technology
Matsugasaki Goshokaido-cho, Sakyo-ku
Kyoto 606-8585, Japan
o-mizuno@kit.ac.jp

ABSTRACT

Comment lines in the software source code include descriptions of codes, usage of codes, copyrights, unused codes, comments, and so on. It is required for comments to explain the content of written code adequately, since the wrong description in the comment may causes further bug and confusion in maintenance.

In this paper, we try to clarify a research question: "In which projects do comments describe the code adequately?" To answer this question, we selected the group 1 of mining challenge and used data obtained from Eclipse and Netbeans. Since it is difficult to answer the above question directly, we define the distance between codes and comments. By utilizing the fault-prone module prediction technique, we can answer the alternative question from the data of two projects. The result shows that Eclipse project has relatively adequate comments.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management

General Terms

Measurement

Keywords

Fault-prone module, comments, source code, distance

1. INTRODUCTION

Comment lines in the software source code include descriptions of code, usage of code, copyrights, unused code, comments, and so on. It is required for comments to explain the content of written code adequately, since the wrong description in the comment may cause further bug and confusion in maintenance. Thus, studies focused on the relationship between comments in source code and software bugs have been done so far [2, 5].

We have studied and developed a fault-prone module prediction approach using text-filtering technique [3]. During the previous works, one remaining issue is clarification of effect of comment lines on the fault-proneness. Can fault-prone modules be predicted using comments only? Which type of comments has significant impact to fault-prone module prediction? Such questions are not yet confirmed.

For the MSR challenge 2011, we try to clarify a research question: "In which projects do comments describe the code adequately?" By answering this question, we can show that which project is well written from the viewpoint of the comment.

To answer the question, we selected the group 1 of mining challenge and used data obtained from Eclipse and Netbeans. Since it is difficult to answer the above question directly, we make an alternative question to answer. To do so, we assumed that the closer the distance between comments and code are, the more adequate described comments are. We then define the measure of distance using the probability to be faulty using the fault-prone module prediction technique. By measuring distance between prediction by comments and prediction by code, we can measure how adequate the comments are.

2. RESEARCH QUESTION

2.1 Question

We try to clarify the following research questions in this challenge:

Question Q In which projects do comments describe the code adequately?

Comments in source code should explain the contents of code adequately. Inadequate comments in source code may cause confusion and may result bugs [5]. We thus try to show how comments reflect the code contents.

However, it is not easy to answer the above question in direct because it is difficult to define adequate comment. We thus make an assumption as follows:

Assumption When we define the distance between comments and code, the closer the distance between comments and code are, the more adequate described comments are.

Since we have proposed text-filtering based approach for fault-prone module prediction [3], we use the approach to measure the distance between code and comments. We can calculate probabilities to be faulty for each comment and code in a module, respectively. Here, we define the distance D between the comments and code as follows:

$$D(M) = |P_{ft}(M_{cmt}) - P_{ft}(M_{code})|, \qquad (1)$$

Table 1: Target Projects

Project	Eclipse	Netbeans
Number of modules	308,966	393,531
Number of faulty modules	159,818 (51.7%)	165,560 (42.1%)
Number or non-faulty modules	149,148 (48.3%)	227,971 (57.9%)
Total lines of code	159,175,979	130,619,502
Comment lines	52,417,180 (32.9%)	36,272,795 (27.8%)
Code lines	106,758,799 (67.1%)	94,346,707 (72.2%)

where M is a source code module, M_{code} and M_{cmt} are code and comment parts in M, respectively, $P_{ft}(M_*)$ is a probability to be faulty for a module calculated by fault-prone filter with given input M_*

By using this measure D, we can answer the question Q by comparing the average values of D between projects. We thus conduct experiments using MSR challenge projects.

2.2 Target Projects

In this study, we use two projects from challenge group 1: Eclipse and Netbeans. Table 1 shows the statistics of both Eclipse and Netbeans projects obtained by SZZ algorithm [4] for these projects. Here, we consider a software module as a class file in Java code.

Both projects are constructed in Java language, and revisions are maintained by concurrent version control system (cvs). The source repository of cvs and bug database used in this study are uploaded one on the MSR challenge Web site. We also used fault reports from the bug database of eclipse project. The type of faults we used in this study is "bugs", therefore these faults do not include any enhancements or functional patches. The status of faults are either "resolved", "verified", or "closed", and the resolution of faults is "fixed". This means that the collected faults have already resolved and fixed and thus fixed revision should be included in the entire repository. The severity of the faults was either blocker, critical, major, or normal. We did not use trivial bugs in this research.

3. FAULT-PRONE FILTERING

3.1 Basic Idea

The basic idea of fault-prone filtering [3] is inspired by spam mail filtering. In spam e-mail filtering, the spam filter first trains both spam and non-spam e-mail messages from the training data set. Then, an incoming e-mail is classified into either spam or non-spam by the filter.

This framework is based on the fact that spam e-mail usually includes particular patterns of words or sentences. From the viewpoint of source code, a similar situation usually occurs in faulty software modules. That is, similar faults may occur in similar contexts. We thus guessed that similar to spam e-mail messages, faulty software modules have similar patterns of words or sentences. To obtain such features, we adopted a spam filter in fault-prone module prediction.

Intuitively speaking, we try to introduce a new metric as a fault-prone predictor. The metric is "frequency of particular words". In detail, we do not treat a single word, but use combinations of words for the prediction. Thus, the frequency of a certain length of words is the only metric used in our approach.

We then try to apply a spam filter to identification of fault-prone modules. We named this approach as "fault-prone filtering". That is, a learner first trains both faulty and non-faulty modules. Then, a new module can be classified into fault-prone or not-fault-prone using a classifier.

3.2 Definition of Comment Lines

Code lines describe a list of operations that developers would like to realize on computers. Comment lines include descriptions of code lines, usage of methods or modules. Code lines are written in a specific programming language, but comment lines are written in a free form.

Previous implementation of fault-prone filtering did not distinguish the comment lines and code lines. This is because the source code module is passed into text filter without any modification. However, since code lines and comment lines have different roles in source code modules, we need to consider such difference in the fault-prone filtering.

For example, comments are usually placed near the difficult code. Therefore, learning the contents of comment lines may be useful to identify the bug-related part in modules. We expect to improve the prediction accuracy of fault-prone filtering by using comment lines effectively.

In this study, we treat a class file as a software module, M. A module M consists of comment lines, M_{cmt}, and code lines, M_{code}. According to the description in Java specification, comment lines into can be separated into classes such as end-of-line comments, block comments, single line comments, training comments, and documentation comments. Among them, the end-of-line comment is a comment that begins with "//" in Java. Since we guess that end-of-line comments have some special effects on source code quality, we use this end-of-line comment class as M_{eol} in further experiment. Note that $M_{eol} \in M_{cmt}$.

3.3 Prediction of Fault-Prone Modules

First of all, inputs of fault-prone filter must be tokenized. Comments are tokenized by the Java syntax scanner as follows:

1. Strings with alphabets, numeric characters, underscore and period
2. Operators in Java language

Other characters, spaces, commas, braces, colons, semicolons, and so on, are used as a separator to generate tokens.

As for the code, we parse and tokenize the code lines using Java parser. Tokens used for prediction is all parsed tokens but separators.

The algorithm of learning and classification is the same as that of [1].

Step 1 Tokens are extracted from faulty and non-faulty modules. The number of occurrence for each token in faulty and non-faulty modules are counted. For a given token t, the following two hash functions can be defined:

- $nonfaulty(t)$: The occurrence of token t in all non-faulty modules.
- $faulty(t)$: The occurrence of token t in all faulty modules.

Step 2 We construct a hash table to obtain the probability, $P_{ft|t}$, by equation (2) that a module that includes the token t is faulty. Here, N_{nf} denotes the number of non-faulty modules

Table 2: Prediction result vs. actual status in Eclipse

(a) PR_{cmt} vs. Actual status

	Prediction by comments	
Actual status	not fault-prone	fault-prone
not faulty	146,078	3,069
faulty	104,647	55,171

Accuracy: 0.651, Precision: 0.947, Recall: 0.345

(b) PR_{eol} vs. Actual status

	Prediction by end-of-line comments	
Actual status	not fault-prone	fault-prone
not faulty	144,175	4,972
faulty	85,838	73,980

Accuracy: 0.706, Precision: 0.937, Recall: 0.463

(c) PR_{code} vs. Actual status

	Prediction by code	
Actual status	not fault-prone	fault-prone
not faulty	139,718	9,429
faulty	40,536	119,282

Accuracy: 0.838, Precision: 0.927, Recall: 0.746

Table 3: Prediction results vs. actual status in Netbeans

(a) PR_{cmt} vs. Actual status

	Prediction by comments	
Actual status	not fault-prone	fault-prone
not faulty	225,998	1,972
faulty	132,620	32,940

Accuracy: 0.658, Precision: 0.944, Recall: 0.199

(b) PR_{eol} vs. Actual status

	Prediction by end-of-line comments	
Actual status	not fault-prone	fault-prone
not faulty	222,073	5,897
faulty	98,691	66,869

Accuracy: 0.734, Precision: 0.919, Recall: 0.404

(c) PR_{code} vs. Actual status

	Prediction by code	
Actual status	not fault-prone	fault-prone
not faulty	210,330	17,640
faulty	38,731	126,829

Accuracy: 0.857, Precision: 0.878, Recall: 0.766

and N_{ft} denotes the number of faulty modules. We call this hash table as a corpus.

$$\begin{aligned} r_{nf} &= \min\left(1, \frac{2 \times nonfaulty(t)}{N_{nf}}\right) \\ r_{ft} &= \min\left(1, \frac{faulty(t)}{N_{ft}}\right) \\ P_{ft|t} &= \max\left(0.01, \min\left(0.99, \frac{r_{ft}}{r_{nf}+r_{ft}}\right)\right) \quad (2) \end{aligned}$$

The classification process is also based on Graham's algorithm.

Step 1 For a new module, M_*, distinct tokens are extracted. In this study, we use all extracted tokens. Here, we count the number of tokens, n, for the next step.

Step 2 According to the equation (3), the probability $P_{ft}(M_*)$ is calculated. We determine whether a module is fault-prone or not. If $P_{ft} \geq 0.9$, the module is determined as fault-prone.

$$P_{ft} = \frac{\prod_{i=1}^{n} P_{ft|t_i}}{\prod_{i=1}^{n} P_{ft|t_i} + \prod_{i=1}^{n}(1 - P_{ft|t_i})} \quad (3)$$

In order to apply our approach to data from source code repository, we implemented tools named "trainer" and "classifier" for training and classifying software modules, respectively. The typical procedure of fault-prone filtering is summarized as follows:

1. Apply classifier to a newly created software module (say, class file in Java, function in C, and so on), M_*^j $(1 \leq j)$, and obtain the probability to be faulty. This step is so-called prediction.

2. By the pre-determined threshold t_{FP} ($t_{FP} = 0.9$ in this study), classify the module M_*^j into either fault-prone or not.

3. When the actual fault-proneness of M_*^j is revealed by fault reports, investigate whether the predicted result for M_*^j was correct or not.

4. If the predicted result was correct, go to step 1; otherwise, apply trainer to M_*^j to learn actual fault-proneness and go to step 1.

This procedure is called "Training only Errors (TOE)" procedure because training process is invoked only when classification errors happen. The TOE procedure is quite similar to actual classification procedure in practice. For example, in actual e-mail filtering, e-mail messages are classified when they arrived. If some of them are misclassified, actual results (spam or non-spam) should be trained.

Intuitively speaking, when we have the ith module to be classfified, we train a model using the 1st to the $i-1$th modules (actually, using a part of them) and classify the ith module.

Hereafter, PR_* denotes a series of predictions using M_* in a project. That is, we have three experiments in a projects, PR_{cmt}, PR_{eol}, and PR_{code}, using M_{cmt}, M_{eol}, and M_{code}, respectively.

3.4 Comparison with Actual Results

Since we have to show that the prediction of fault-proneness by text filtering is valid, we investigate whether the results of prediction match to the actual results. Tables 2 and 3 show the result of fault-prone module prediction using comments in Eclipse and Netbeans, respectively.

In Table 2(c), we can see that PR_{code} has the best prediction result of all because it has the highest accuracy and relatively high precision and recall. On the other hand, in Table 2(a), PR_{cmt} shows lower accuracy and especially, it has the lowest recall of all. This implies that PR_{cmt} is confused by contents of comments and results inaccurate prediction. However, we can find that PR_{eol} has marginal results between PR_{code} and PR_{cmt} in Table 2(b). This fact implies that end-of-line comments have rather good information for fault-proneness prediction.

We can also see that the above mentioned trend resembles between two projects as shown in Table 2 and Table 3.

4. COMPARISON BETWEEN PROJECTS

As we stated in Section 2, we try to show whether comments of source code adequately describe the content of code. To do so, we first compare the predicted results by comments and by code for

Table 4: Investigation of Prediction Distance in Eclipse

(a) PR_{cmt} vs. PR_{code}

	Prediction by comments	
Prediction by code	not fault-prone	fault-prone
not fault-prone	175,021	5,234
fault-prone	75,705	53,006

Correspondence: 0.738, Average D^2: 0.256

(b) PR_{eol} vs. PR_{code}

	Prediction by end-of-line comments	
Prediction by code	not fault-prone	fault-prone
not fault-prone	172,187	8,068
fault-prone	57,827	70,884

Correspondence: 0.787, Average D^2: 0.195

Table 5: Investigation of Prediction Distance in Netbeans

(a) PR_{cmt} vs. PR_{code}

	Prediction by comments	
Prediction by code	not fault-prone	fault-prone
not fault-prone	246,206	2,856
fault-prone	112,413	32,056

Correspondence: 0.707, Average D^2: 0.290

(b) PR_{eol} vs. PR_{code}

	Prediction by end-of-line comments	
Prediction by code	not fault-prone	fault-prone
not fault-prone	242,858	6,204
fault-prone	77,907	66,562

Correspondence: 0.786, Average D^2: 0.198

both Eclipse and Netbeans. We then calculate the distance D in equation (1) of a prediction for each experiment. In order to compare the absolute amount of distance, we show the average value of D^2 for each experiment.

The results of comparison are shown in Tables 4 and 5. The result in Eclipse project is shown in Table 4. In Table 4(a), the results between prediction by comments and prediction by code are shown. Here, "correspondence" indicates the rate of the same predictions between two methods. "Average D^2" shows the average distance between two methods in equation 1. Table 4(a) shows the result between prediction by end-of-line comments and prediction by code. From Table 4, we can see that the result of prediction by end-of-line comments is closer to that of code than comments because it has small average D^2 and large correspondence.

The result in Netbeans project is shown in Table 5. Let us compare Table 4(a) and 5(a). We can see that Eclipse project has smaller D^2 and larger correspondence.

The above mentioned trend holds when we compare Table 4(b) and 5(b). However, the difference between projects becomes very small.

Consequently, we can interpret this result as follows:

1. When we focus on comments, M_{cmt} in Eclipse is more adequately described than M_{cmt} in Netbeans.

2. On the other hand, when we focus on end-of-line comments, M_{eol} in Eclipse is as adequate as M_{eol} in Netbeans.

The answer to the research question Q is, "Eclipse has more adequate comments than Netbeans, but there is no difference between Eclipse and Netbeans if we focus on end-of-lines comments".

5. THREATS TO VALIDITY

External validity mainly includes the generalizability of the proposed approach. In this study, we have a certain external validity threats since we investigated only two projects from Group 1 of MSR challenge 2011. Because these projects are the same type (IDEs), it is possible to show similar trends. We have to investigate more projects in future research.

As for internal validity, the definition of distance between code and comments is a threat. At this point, we do not have alternative idea to define the distance. The training only errors approach can be another threat. This procedure tends to show higher prediction results since there are many almost the same modules in the software history. In the future research, we have to find more rigid approach to evaluate prediction result.

One of the construction validity threats is the collection of fault-prone modules from open source software projects. The number of faults found in the repository was not large with respect to that in a bug database. The algorithm adopted in this study has the limitation that faults not recorded in the repository log cannot be collected. To make an accurate collection of faulty modules from the source code repository, further research is required.

The way of statistical analysis usually causes threats to conclusion validity. We cannot find any threats to conclusion validity in our study at this point.

6. CONCLUSIONS

In this paper, we tried to investigate the possibility of comments for fault-prone prediction. For this purpose, we aim to answer the question, "Do comments describe the code adequately?" We conducted experiments using Eclipse and Netbeans data sets and showed that (1) when we focus on comments, comments in Eclipse is more adequately described than in Netbeans, and (2) when we focus on end-of-line comments, end-of-lines comments in Eclipse is as adequate as in Netbeans.

7. ACKNOWLEDGMENTS

The authors would like to thank Mr. Hideaki Hata, PhD student in Osaka University, who gave us invaluable comments for this study. This research is partially supported by the Japan Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 20700025, 2008–2010.

8. REFERENCES

[1] P. Graham. *Hackers and Painters: Big Ideas from the Computer Age*, chapter 8, pages 121–129. O'Reilly Media, 2004.

[2] R. K. Lind and K. Vairavan. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Trans. Softw. Eng.*, 15(5):649–653, 1989.

[3] O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *Proc. of 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 405–414, 2007.

[4] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? (on Fridays.). In *Proc. of 2nd International workshop on Mining software repositories*, pages 24–28, 2005.

[5] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: bugs or bad comments? */. In *Proc. of SOSP'07*, 10 2007.