LETTER

Change-Prone Java Method Prediction by Focusing on Individual Differences in Comment Density*

Aji ERY BURHANDENNY[†,††a)], *Nonmember*, Hirohisa AMAN[†††], *and* Minoru KAWAHARA[†††], *Members*

SUMMARY This paper focuses on differences in comment densities among individual programmers, and proposes to adjust the conventional code complexity metric (the cyclomatic complexity) by using the abnormality of the comment density. An empirical study with nine popular open source Java products (including 103,246 methods) shows that the proposed metric performs better than the conventional one in predicting change-prone methods; the proposed metric improves the area under the ROC curve (AUC) by about 3.4% on average.

key words: *change-prone method, comment density, individual difference, ROC curve, AUC*

1. Introduction

Comments are both well-known artifacts for enhancing the readability of programs and useful embedded documents [1]. They can provide various information including the copyright designation, the programmers' memos on the code fragments and the manuals of their functions [2]. While comments are independent of program executions, we cannot ignore the impact of comments on the code quality due to their helpful effects on the program comprehension. However, there have also been concerns that some programmers might add detailed comments to their complicated code in order to compensate for a lack of readability [3]; In the code refactoring world, well-written comments are said to be "deodorant" for masking code smells [4]. There are empirical reports showing that the comments within a Java method body—hereinafter referred to as "inner comments"—are noteworthy artifacts in analyzing the code quality, and more-commented methods are more likely to be fixed after their releases (i.e., change-prone) [5]–[7].

However, a programmer's preference may also play an important role in commenting source code. While one programmer prefers to write detailed comments, another programmer does not like to write comments at all. If the latter programmer wrote a well-commented method, it would be an abnormal case and we should review the method preferentially. That is to say, such an abnormality depends on who wrote the program. The aim of this paper is to evaluate the abnormality of a method from the perspective of the comment density with consideration for differences among individual programmers, and to empirically examine if such an abnormality can help in a change-prone method prediction.

Manuscript received November 21, 2016.
Manuscript revised January 28, 2017.
Manuscript publicized February 15, 2017.

[†]The author is with Graduate School of Science and Engineering, Ehime University, Matsuyama-shi, 790–8577 Japan.
[††]The author is with Engineering Faculty, Mulawarman University, Samarinda, East Kalimantan, 75119 Indonesia.
[†††]The authors are with Center for Information Technology, Ehime University, Matsuyama-shi, 790–8577 Japan.

[*]The basic idea of this paper was presented at the 2016 International Conference on Business and Information (BAI2016) [20].

a) E-mail: aji@se.cite.ehime-u.ac.jp
DOI: 10.1587/transinf.2016EDL8224

2. Comment Density Considering Individual Differences and Its Application to Complexity Metric

In general, more complex methods tend to be harder to understand, and may have more inner comments than simpler ones. To evaluate the amount of comments regardless of the complexity, we focus on the comment density, the lines of inner comments normalized by the complexity. In this paper, we will use the cyclomatic complexity [8], which has been a well-known complexity metric, as our complexity criterion.

In order to consider individual differences in commenting code, we need to make a link between a method and its programmer. In general, two or more programmers may be involved in a method development through its maintenance. Thus, we focus on the initial version of method because we can always determine a specific programmer for each method; if we focus on later versions of methods which two or more developers have made modifications, it is hard to evaluate the amount of comments for each developer. A further analysis taking care of multi-developer methods is our significant future work.

Now, let N be the number of programmers, and denote them by p_i (for $i = 1, \ldots, N$). Suppose p_i has developed M_i methods, m_{ij} (for $j = 1, \ldots, M_i$). Then, we define the comment density of method m_{ij} "CD(m_{ij})" as follows:

$$\mathrm{CD}(m_{ij}) := \log\left\{ \frac{\mathrm{LCM}(m_{ij})}{\mathrm{CC}(m_{ij})} + 1 \right\} \tag{1}$$

where LCM(m_{ij}) and CC(m_{ij}) are the lines of inner comments of m_{ij} and the complexity of m_{ij}, respectively. While a comment density can also be obtained by the simple ratio "LCM(m_{ij})/CC(m_{ij})," the distribution of such values is likely to be right-skewed, so we will use a logarithmically transformed form shown in Eq. (1)[**].

[**]Since LCM(m_{ij}) can be zero, we inserted "+1" into the equation: if LCM(m_{ij}) = 0, then CD(m_{ij}) = log(0 + 1) = 0.

Copyright © 2017 The Institute of Electronics, Information and Communication Engineers

Based on $\mathrm{CD}(m_{ij})$, we define the comment density considering individual difference, "$\mathrm{zCD}(m_{ij})$," as follows:

$$\mathrm{zCD}(m_{ij}) = \frac{\mathrm{CD}(m_{ij}) - \mu_i}{\sigma_i} \qquad (2)$$

where μ_i and σ_i are the mean and the standard deviation of programmer p_i's comment densities $\mathrm{CD}(m_{ij})$ (for $j = 1, \ldots, M_i$), respectively.

$\mathrm{zCD}(m_{ij})$ is a standard score of $\mathrm{CD}(m_{ij})$, which is referred to as "z-score." A larger value of $\mathrm{zCD}(m_{ij})$ shows that m_{ij} has a higher comment density than programmer p_i's usual work. Since its scale is normalized by the dispersion (σ_i), a high zCD value signifies that the comment density is abnormally high for the programmer. That is to say, zCD can be a metric for measuring an abnormality of method from the perspective of the comment density.

Now we make a hypothesis that zCD is useful in finding problematic methods. Needless to say, the conventional complexity metric, CC, is a promising metric for evaluating programs. Then, we consider an enhancement of CC by using zCD, and define the following Adjusted CC, "ACC":

$$\mathrm{ACC}(m_{ij}) = \phi(m_{ij}) \cdot \mathrm{CC}(m_{ij}) \qquad (3)$$

where $\phi(m_{ij})$ is the degree of attention to m_{ij}. In order to calculate $\phi(m_{ij})$ based on the abnormality of m_{ij}, we propose to use the cumulative normal distribution function as follows:

$$\begin{aligned}\phi(m_{ij}) &= \Pr\left\{\, x \le \mathrm{zCD}(m_{ij}) \,\right\} \\ &= \int_{-\infty}^{\mathrm{zCD}(m_{ij})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \ . \qquad (4)\end{aligned}$$

The range of $\phi(m_{ij})$ is $0 < \phi(m_{ij}) < 1$. When a method has an average comment density, $\phi(m_{ij})$ is around 0.5. When $\mathrm{zCR}(m_{ij})$ is higher—an abnormally-high comment density—, $\phi(m_{ij})$ gets closer to 1; when $\mathrm{zCR}(m_{ij})$ is lower, $\phi(m_{ij})$ approaches 0.

3. Empirical Study

3.1 Aim and Dataset

The aim of this empirical study is to quantitatively examine if ACC can be useful for detecting change-prone Java methods. Since change-prone methods cannot survive unscathed after their releases, they should be reviewed more carefully.

We collected methods from nine popular open source software (OSS) projects shown in Table 1. The main reasons why we selected those nine projects as our data source are: (a) they are popular OSS projects, (b) their source files have been maintained with the Git, and (c) they are written in Java.

For the reason (a), we believe that it is better to use popular projects; results derived from minor projects would be worthless. All of these nine projects are ranked in the top 50 popular Java projects at SourceForge.net[†]. The reason (b) is for an ease of data collection. The Git provides powerful functions to collect data including source code and their changes. The reason (c) is from our data collection tool[††].

[†]https://sourceforge.net/

Table 1 Surveyed OSS projects.

project name	#examined methods
Angry IP Scanner [9]	1,669
Eclipse Checkstyle Plugin [10]	1,511
eXo Platform [11]	1,362
FreeMind [12]	6,920
GNU ARM Eclipse Plug-ins [13]	3,813
Hibernate ORM [14]	34,395
PMD [15]	2,510
ProjectLibre [16]	30,090
SQuirreL SQL Client [17]	20,946
total	103,246

For each source file, we traced all change logs and found the initial version of each method. Then, for each method, we collected the following data: (1) CC value, (2) LCM value, (3) the author's name and e-mail address, and (4) the number of changes that occurred after the release. We compute CD value for each method with Eq. (1), and get the mean (μ_i) and the standard deviation (σ_i) of the CD values for each programmer. Then, we compute zCD value for each method with Eq. (2). Finally, we obtain ACC values with Eqs. (3), (4).

There may be an author who has two or more different names or e-mail addresses on the repository. We tried integrating duplicated author data by the following set of rules—it is a simpler version of the algorithm proposed by Bird et al. [18]: (1) if two authors have different addresses but the same name, then we regard them as the same author; (2) if two authors have the same address but different names, then we regard them as the same author.

3.2 Assessment Criterion

To evaluate the performance of change-prone method prediction using ACC, we leverage the receiver operating characteristic (ROC) curve [19]. When a method's ACC value is greater than an established threshold, we judge the method is change-prone. Then, we can see the false-positive (FP) rate and the false-negative (FN) rate. The ROC curve shows the relationships between FP rate and FN rate for all available thresholds. The area under the curve (AUC) is a well-known criterion of the performance—a higher AUC value signifies a better performance in discriminating change-prone methods. We will compare the AUC values of ACC (the proposed metric) and the ones of CC (the conventional metric).

Since the number of code changes that occurred in most of methods is less than two (see Table 2), we consider the methods which had been modified two or more times after their release, to be change-prone methods in this paper.

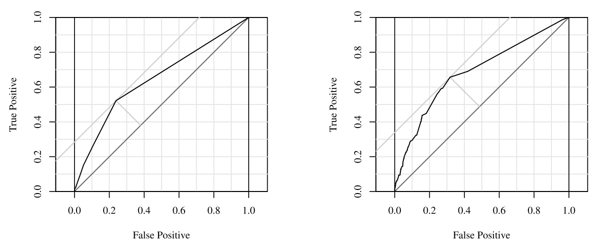
3.3 Results and Discussions

Figure 1 shows ROC curves for the Angry IP Scanner: (a)

[††]http://se.cite.ehime-u.ac.jp/tool/

Table 2 Distribution of code change counts in methods.

project name	min	25%	50%	75%	max
Angry IP Scanner	0	0	0	1	45
Eclipse Checkstyle Plugin	0	0	0	1	7
eXo Platform	0	0	0	1	14
FreeMind	0	0	1	1	61
GNU ARM Eclipse Plug-ins	0	0	0	1	33
Hibernate ORM	0	0	0	1	47
PMD	0	0	0	1	13
ProjectLibre	0	0	0	0	7
SQuirreL SQL Client	0	0	0	0	21



(a) conventional metric (CC) (b) proposed metric (ACC)

Fig. 1 ROC curves for "Angry IP Scanner."

Table 3 AUC values: CC vs. ACC.

	AUC (α)		
project name	CC	ACC	δ
Angry IP Scanner	0.647	0.685	5.922
Eclipse Checkstyle Plugin	0.672	0.692	2.905
eXo Platform	0.690	0.718	4.068
FreeMind	0.665	0.691	3.940
GNU ARM Eclipse Plug-ins	0.604	0.632	4.549
Hibernate ORM	0.665	0.678	1.918
PMD	0.600	0.624	3.996
ProjectLibre	0.819	0.852	4.010
SQuirreL SQL Client	0.602	0.598	−0.700
average	—	—	3.401

and (b) are the ROC curves obtained by using the conventional metric (CC) and the proposed metric (ACC), respectively. Because of space limitations, we will omit the ROC curves for the remaining eight projects. Table 3 presents the AUC values for all projects. δ in the table denotes the rate of improvement by ACC instead of CC, defined as follows:

$$\delta = 100 \cdot \frac{\alpha(\mathrm{ACC}) - \alpha(\mathrm{CC})}{\alpha(\mathrm{CC})} \quad (\%), \tag{5}$$

where α(CC) and α(ACC) are the AUC values of ROC curves obtained by using CC and ACC, respectively.

All projects except for the SQuirreL SQL Client show $\delta > 0$ (see Table 3), which means the proposed metric performs better than the conventional one in terms of change-prone method prediction. Although the SQuirreL gives a negative rate, it is almost zero (−0.7%), so the proposed metric is at almost the same level of performance as the conventional metric for the SQuirreL. In total, the average value of δ is 3.4%. While the improvement made by using ACC is not especial high, it showed better performances for eight out of nine sampled projects. Therefore, an abnormality of comment density seems to be worthy of attention.

As an ACC value is a CC value multiplied by an attention degree which is in the range between 0 and 1, it is highly unlikely that an ACC value of a method differs drastically from the CC value of the method. This would be one of the major reasons why the metrics ACC and CC show similar performances. Nonetheless, the actual results show that ACC performs better than CC for eight out of nine projects. Since the multiplication of the attention degree can change the order of methods in their metric values, the adjustment seems to work successfully in detecting change-prone methods. That is to say, the abnormality of comment density by programmer would be a useful factor for enhancing the change-prone method prediction.

3.4 Threats to Validity

Our results may be subject to the following concerns.

We investigated nine OSS projects and the number of subjects might be considered small, so there is a threat of generalization. However, since the our analysis unit is a "method" and not project, the above threat would be mitigated.

We analyzed OSS projects only written in Java. While programs written in another language might show different results, the notion of comment description is common to most of the modern programming languages. Thus, we believe the limitation of programming language cannot be a serious threat to validity in this paper.

Our data of methods are of their "initial versions" but not the ones of a certain release version. Therefore, our results might not work well for predicting code changes after the release of interest. As we mentioned in Sect. 2, there is the issue of "multi-developer" of methods. In this paper, we proposed to evaluate an abnormality of comments by focusing on each developer. In order to take into account a method which has been developed and maintained by two or more developers, we have to make another definition of abnormality in commenting, i.e., a multi-developer version of it. Since such multi-developer cases should be addressed as well, we plan to perform further analyses in order to produce a reasonable definition in the future.

4. Conclusion

We proposed to adjust the code complexity metric by using the abnormality of comment densities. Since there can be a wide variety in commenting code among programmers, the abnormality of comment densities was considered in this paper. Through our empirical study with nine OSS projects, we showed that the proposed metric works better than the conventional complexity metric in predicting change-prone Java methods which cannot survive unscathed after their releases. Our future work includes a further analysis focusing on not only comment density but also their contents, and an extension of our definition to multi-developer cases.

Acknowledgments

We would like to thank the anonymous reviewer for the helpful comments to the earlier version of this paper. This work was supported by JSPS KAKENHI Grant #16K00099 and DIKTI Scholarships, Directorate Generale of higher Education of Indonesia.

References

[1] S.C.B. de Souza, N. Anquetil, and K.M. de Oliveira, “A study of the documentation essential to software maintenance,” Proc. 23rd Int’l Conf. Design of Communication, pp.68–75, Sept. 2005.
[2] D. Steidl, B. Hummel, and E. Juergens, “Quality analysis of source code comments,” Proc. 21st Int’l Conf. Program Comprehension, pp.83–92, May 2013.
[3] R.P.L. Buse and W.R. Weimer, “A metric for software readability,” Proc. 2008 Int’l Symposium on Softw. Testing & Analysis, pp.121–130, July 2008.
[4] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman, Boston, MA, 1999.
[5] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, “Lines of comments as a noteworthy metric for analyzing fault-proneness in methods,” IEICE Trans. Inf. & Syst., vol.E98-D, no.12, pp.2218–2228, Dec. 2015.
[6] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, “Empirical analysis of fault-proneness in methods by focusing on their comment lines,” Proc. 21st Asia-Pacific Softw. Eng. Conf., vol.2, pp.51–56, Dec. 2014.
[7] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, “Empirical analysis of change-proneness in methods having local variables with long names and comments,” 2015 ACM/IEEE Int’l Symp. Empirical Softw. Eng. & Measurement, pp.50–53, Oct. 2015.
[8] T.J. McCabe, “A complexity measure,” IEEE Trans. Softw. Eng., vol.SE-2, no.4, pp.308–320, Dec. 1976.
[9] “Angry IP Scanner.” http://angryip.org/, accessed Aug. 26. 2016.
[10] “Eclipse Checkstyle Plugin.” http://eclipse-cs.sourceforge.net/, accessed Aug. 26. 2016.
[11] “eXo Platform.” https://www.exoplatform.com/, accessed Aug. 26. 2016.
[12] “FreeMind.” http://freemind.sourceforge.net/, accessed Aug. 26. 2016.
[13] “GNU ARM Eclipse Plug-ins.” http://gnuarmeclipse.github.io/, accessed Aug. 26. 2016.
[14] “Hibernate ORM.” http://hibernate.org/orm/, accessed Aug. 26. 2016.
[15] “PMD.” https://pmd.github.io/, accessed Aug. 26. 2016.
[16] “ProjectLibre.” http://www.projectlibre.org/, accessed Aug. 26. 2016.
[17] “SQuirreL SQL Client.” http://squirrel-sql.sourceforge.net/, accessed Aug. 26. 2016.
[18] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, “Mining email social networks,” Proc. 2006 Int’l Workshop Mining Softw. Repositories, pp.137–143, May 2006.
[19] A. Agresti, Categorical Data Analysis, Wiley Interscience, N.J., 2002.
[20] A.E. Burhandenny, T. Nakano, H. Aman, and M. Kawahara, “Empirical study of change-prone and fault-prone method prediction focusing on comment ownership,” Proc. 2016 Int’l Conf. Business & Inf., pp.219–230, July 2016.