Empirical Analysis of Change-Proneness in Methods Having Local Variables with Long Names and Comments

Hirohisa Aman
Center for Information Technology
Ehime University
Matsuyama, Japan 790–8577
Email: aman@ehime-u.ac.jp

Sousuke Amasaki
Faculty of Computer Science
and Systems Engineering
Okayama Prefectural University
Soja, Japan 719–1197

Takashi Sasaki and Minoru Kawahara
Center for Information Technology
Ehime University
Matsuyama, Japan 790–8577

***Abstract*—This paper focuses on the local variable names and comments that are major artifacts reflecting the programmer's preference. It conducts an empirical analysis on the usefulness of those artifacts in assessing the software quality from the perspective of change-proneness in Java methods developed in six popular open source software products. The empirical results show: (1) a method having a longer named local variable is more change-prone, and (2) the presence of comments inside the method body strengthens the suspicions to be modified after the release. The above artifacts are worthy to find methods which can survive unscathed after the release.**

I. Introduction

Programming is one of the most crucial activities in software development. While well-done upper processes, i.e., successful requirements analysis and system/module design are required to develop high quality software products, the importance of those activities are based on proper programming (implementations). Since programming activities are usually done by programmers, i.e., human beings, the product quality can vary from person to person. That is to say, different programmers may write different programs for the same specification: one programmer writes a simple and comprehensible program, but another programmer writes a complicated program that is hard to understand. Thus, it would be significant to focus on the factors reflecting the programmer's preference for successful quality management. We will look at two types of artifacts on which programmers have considerable discretion—"comments" and "local variables."

Comments written in a program have no impact on the program's execution. It is at the programmer's discretion whether they write comments or not, and what they describe as comments. However, comments do not seem to be independent of program quality: well-written comments have been said to be related to "code smells" of problematic code fragments that should be refactored [1]. Although comments themselves do not have harmful effects on the program, some programmers write detailed comments to compensate for a lack of clearness in their complicated code fragments. That is to say, comments may be used as "deodorant" masking a code smell, and they can be signs to detect parts of poor quality [2], [3], [4], [5].

Local variables declared in a method or a function are also artifacts that the programmer would be able to decide freely. Although names of other variables—arguments of a method/function and global variables—are often controlled at the preceding design phase, names of local variables would be at the programmer's discretion. There would be diversity in naming—one programmer prefers short names such as `i` and `c` (single letter) or `idx` and `cnt` (abbreviated word), but another programmer may like to use long and meaningful names such as `index` and `countOfCollisions` (fully spelled word). While a variable having a longer name can provide richer information and can be easier to understand, long named variables may not be often required in a narrow scope [6]. Some coding standards [7], [8] also say that local variable names should (can) be shorter. If a programmer dared to use a long named local variable in a method/function, it might be a sign that the programmer considered the code fragment was complicated. It may be yet another code smell—this is the motivation of our research.

The contribution of this paper is to report the results of our empirical analysis on the chances of modifications or bug fixes occurred in methods having long named local variables and comments. To the best of our knowledge, there has not been an empirical study on change-proneness (including fault-proneness) by focusing on both comments and the lengths of local variable names in the past.

The remainder of this paper is organized as follows. Section II describes the related work. Section III presents an empirical analysis for six popular open source software products. Finally, Section IV gives the conclusion and our future work.

II. Related Work

It has been widely known that comments are useful artifacts to enhance the comprehensibility of programs [9], [10], [11]. However, comments are sometimes added to compensate for a lack of clearness in the program—such a program is hard to understand without its comments. Kernighan et al. stated that you should rewrite your program if you wanted to add detailed comments [6]. Fowler pointed out that well-written comments may be a deodorant for a code smell [1]. While comments are harmless to programs, they may be signs of problematic code fragments. Aman et al. focused on those relationships between comments and problematic programs, and conducted empirical analyses on the fault-proneness in

commented programs [2], [3], [4], [5]. Their empirical results showed that the presence of comments written inside method bodies (they called "inner comments") is correlated with the fault-proneness in the programs. Therefore, inner comments may be useful clues of code fragments to be revised. Steidl et al. proposed a refactoring manner to extract the code fragments having one-line comments as a new method [12].

Lawrie et al. conducted a survey on program comprehension by focusing on the difference in style of variable names: (1) single letters, (2) abbreviated words and (3) fully spelled words such as (1) `i`, (2) `idx` and (3) `index`, respectively [13]. Their survey results showed that the comprehension level of the programs were monotonically increased as the variable name got longer from (1) to (3), but there was no significant difference between (2) abbreviated words and (3) fully spelled words. Thus, it seems that a long named local variable is not always needed, and similar things are also said in the GNU coding standard [7] and Java Coding Style [8] as local variable names should (can) be shorter. The survey results by Lawrie et al. and the heuristics in the coding standards motivated our empirical analysis presented in the following section.

III. Empirical Analysis

This section presents our empirical analysis on change-proneness in Java methods by focusing on the presence of long named local variables and comments. Since a method having a problematic code fragment requires some modifications or bug fixes after its first release, the change-proneness may be a simple valuable measure for assessing the method's quality and maturity level.

A. Research Questions

To clarify our goal in this empirical analysis, we consider the following two research questions (RQs).

RQ1: Is the presence of long named local variables related to change-proneness?

This research question is to empirically examine the heuristics that shorter names are better for local variables. If the answer to RQ1 is YES, the length of a local variable is considered to be a simple but useful metric for assessing code quality.

RQ2: Does the presence of inner comments affect the answer to RQ1?

While the previous work [2], [3], [4], [5] reported that the presence of inner comments is related to fault-proneness in programs, there has not been a study on the combination of comments and local variables. The examination of this combination can clarify the effects of two factors, comments and local variables.

B. Datasets

We analyzed six popular Open Source Software (OSS) products of different size and domain, shown in Table I—Angry IP Scanner (IP-Scanner)[1], Eclipse Checkstyle Plug-in (Checkstyle)[2], FreeMind[3], GNU ARM Eclipse Plug-ins (ARM)[4], Hibernate ORM (Hibernate)[5] and SQuirreL SQL Client (SQuirreL)[6]. They all ranked in the top 50 popular Java products at SourceForge.net[7], are developed in Java, and their source files are maintained with Git. The restrictions of language and version control systems are from our data collection tools[8].

TABLE I. OSS products analyzed in this paper.

Product	KLOC	Data Collection Period	Domain
IP-Scanner	16	2006-07-19 — 2015-06-05	Networking
Checkstyle	21	2003-05-05 — 2015-04-14	Code analysis
FreeMind	72	2011-02-06 — 2014-06-28	Mind-mapping tool
ARM	266	2013-09-11 — 2015-04-06	Development support
Hibernate	528	2007-06-29 — 2015-05-11	Object/Relational mapping
SQuirreL	397	2001-06-01 — 2015-05-09	Database client
Total	1,300		

We conducted the following data collection scheme for each OSS product at the end of May, 2015.

1) Make the latest list of methods.
Make a clone of the latest Git repository, and list the methods (including constructors) which appear in the source files. Abstract methods and methods for testing or demos are excluded from our method list.

2) Track the changes of each method.
For each method, track the changes of source files including the method, then check whether the method has been modified or not. Specifically, (1) make a list of commitments involved in the source file, (2) check out the different versions of the file corresponding to consecutive commitments, (3) extract the method body from each version of each source file, and (4) decide whether the method was modified or not by comparing the content of two methods corresponding to consecutive versions. We regard a modification as one for fixing a bug if bug-fixing-related keywords or bug IDs appeared in the commitment log [14]

3) Measure metric values for the initial version of each method.
For the initial version of each method, measure Lines of Inner Comments (LIC)—the number of comment lines in the method body— and the lengths of local variable names. Measure the length of name from two different perspectives: the letter count and the word count. Our measurement of the word count is based on the camel case which is normally used in Java, in which the word count is the number of lower-case letters followed by an upper-case letter, plus one. ■

Table II shows the number of methods collected, the ratios of one or more change occurrences, two or more change occurrences, and bug fix occurrences in the methods of each OSS product. While there are individual differences in the number of methods and the ratios of change or bug fix occurrences, we believe it is important to find a commonly-observed tendency for such data with diversity. Tables III and IV show summarized statistics of the lines of inner comments and the length of local variable names, respectively. The majority of methods have no inner comments or no local variable. The

[1] http://angryip.org/
[2] http://eclipse-cs.sourceforge.net/
[3] http://freemind.sourceforge.net/wiki/index.php/Main_Page
[4] http://gnuarmeclipse.livius.net/blog/
[5] http://hibernate.org/
[6] http://www.squirrelsql.org/
[7] http://sourceforge.net/
[8] http://se.cite.ehime-u.ac.jp/tool/

TABLE II. NUMBER OF METHODS IN EACH PRODUCT AND THE OCCURRENCES OF CHANGES AND BUG FIXES AT THOSE METHODS.

Product	#methods	Occurrence (ratio)		
		One or more times changes	Two or more times changes	Bug fix
IP-Scanner	1,504	27.3%	13.6%	4.3%
Checkstyle	1,593	30.6%	10.6%	19.5%
FreeMind	7,493	35.1%	8.3%	31.0%
ARM	3,729	39.6%	17.4%	4.3%
Hibernate	25,023	27.3%	7.9%	15.5%
SQuirreL	22,319	16.1%	6.0%	6.9%
Total	61,661	25.0%	8.1%	13.4%

TABLE III. RATIO OF NON-COMMENTED METHODS AND DISTRIBUTION OF LINES OF INNER COMMENTS(LIC).

Product	Ratio of LIC = 0	Distribution of LIC except for LIC= 0				
		Min.	Q_1	Q_2	Q_3	Max.
IP-Scanner	77.2%	1	1	1	2	17
Checkstyle	69.9%	1	1	2	3	130
FreeMind	83.2%	1	1	2	3	351
ARM	75.4%	1	1	2	4	79
Hibernate	88.9%	1	1	2	3	72
SQuirreL	89.2%	1	1	2	4	130
Total	86.7%	1	1	2	3	351

Q_1: 25 percentile; Q_2: median; Q_3: 75 percentile

TABLE IV. RATIO OF METHODS HAVING NO LOCAL VARIABLE AND DISTRIBUTION OF LETTER COUNTS OF THE LONGEST-NAME LOCAL VARIABLES.

Product	Ratio of no variable	Distribution of maximum letter counts				
		Min.	Q_1	Q_2	Q_3	Max.
IP-Scanner	72.2%	1	5	8	12	25
Checkstyle	53.7%	1	6	10	13	38
FreeMind	68.6%	1	5	9	13	43
ARM	65.4%	1	5	9	13	33
Hibernate	77.3%	1	5	9	14	42
SQuirreL	72.5%	1	5	8	12	40
Total	73.1%	1	5	9	13	43

Q_1: 25 percentile; Q_2: median; Q_3: 75 percentile

distribution of a variable name's length is approximately the same. Since the longest named variables in terms of "letter count" were also the longest named ones in terms of "word count" in 96.6% of methods, we showed only "letter count" data in the table; we will deal with the length of local variable using the letter count in the following analysis.

C. Analysis for RQ1: Is the presence of long named local variables related to change-proneness?

We compared the ratios of change occurrences and bug fix occurrences among the categories of methods shown in Table V. These categories are based on the length of the variable's name. Single letter or two-letter variables are considered to be the category of methods which have only much shorter variables (C_1). Since three or more letters may form an abbreviated word or a full-spelled word, we regarded three(3) letters as a threshold between C_1 and C_2. Nine(9) letters is the median of all variable names (see Table IV), so we separated C_3 from C_2 by setting nine as the upper limit of C_2.

Figure 1 shows the ratios of one or more change occurrences, two or more change occurrences, and bug fix occurrences for each OSS product. There seem to be roughly increasing tendencies for all ratios in all products. We identified that those increasing tendencies are statistically significant through the Cochran-Armitage test [15] at a 0.01 significance level. Thus, it seems to be a common tendency that a method having a longer named local variable is more change-prone. Such a method has a low probability to survive unscathed, and it would require more costs to be improved after the release.

TABLE V. METHOD CATEGORIES.

category	description
C_0	methods having no local variable
C_1	methods whose longest named local variable consists of one or two letters
C_2	methods whose longest named local variable consists of n letters, where $3 \leq n \leq 9$
C_3	methods whose longest named local variable consists of more than 9 letters

Since the larger method is more change-prone and more fault-prone in general, there was a concern that the method size (lines of code: LOC) might be a confounding factor in our results. We computed Spearman's rank-correlation coefficients between LOC and the length of variable names for the methods having local variables. The coefficient resulted in 0.352 for all data; for each OSS product, the coefficients were between 0.290 and 0.468. That is to say, there seem to be only weak correlations. Therefore, we can say the presence of long named local variables is related to change-proneness.

D. Analysis for RQ2: Does the presence of inner comments affect the answer to RQ1?

In order to examine the effects of inner comments with long named local variables, we divided each category in RQ1 into two sub categories corresponding to commented methods and non-commented methods. Figure 2 shows the differences of ratios between commented methods and non-commented methods: the horizontal axes correspond to categories C_0, C_1, C_2 and C_3, and the vertical axes signify the ratio of commented methods minus the ratio of non-commented methods, respectively. All differences of ratios except for C_2 in Checkstyle are positive. That is to say, commented methods have higher change ratios and higher bug fix ratios than non-commented ones. Only the results of C_2 in Checkstyle differed from the others (see Fig.2(b)), and it also showed different tendencies in RQ1 (see Fig.1(b)). Since the ratio of commented methods in Checkstyle are higher than the other OSS products (the lowest ratio of LIC= 0 in Table III), the product might have a slightly different commenting manner from the others.

Although the differences in ratios were not always statistically significant[9], we saw a tendency that commented methods are more change-prone in most cases of the results, and the presence of comments does not invalidate the results of RQ1. Rather, the usefulness of the results in RQ1 can be enhanced by focusing on the presence of comments.

From the empirical results, we suggest to practitioners that they conduct a review of their programs when they wanted to declare local variables with longer names and to add inner comments; this can be yet another code smell.

[9]We performed statistical tests on the differences in ratios for 12 pairs (4 categories × 3 types of ratios). The following pairs showed significant differences at an 1% significance level: 3/12 in IP-Scanner, 2/12 in Checkstyle, 11/12 in FreeMind, 8/12 in ARM, 10/12 in Hibernate and 2/12 in SQuirreL.

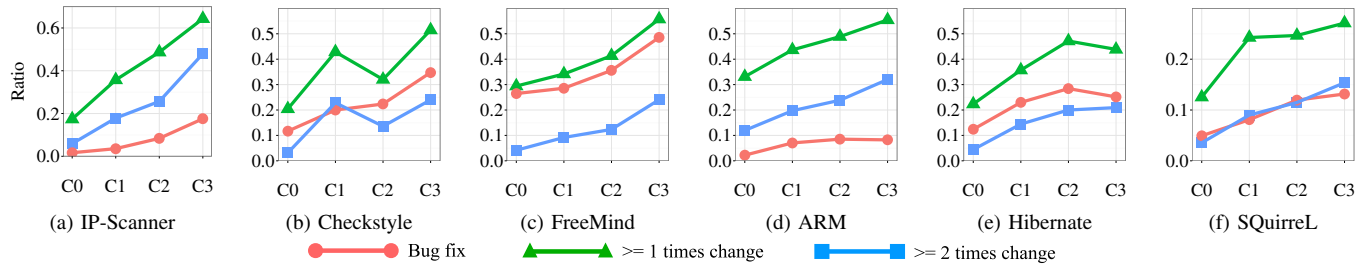

Fig. 1. Change ratios and bug fix ratios of methods in each OSS product.

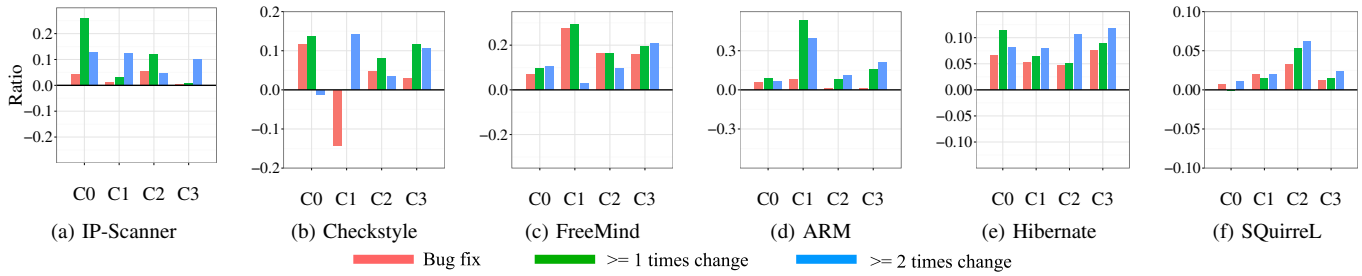

Fig. 2. Differences in change ratio and bug fix ratio between commented methods and non-commented methods.

E. Threats to Validity

In general, there is diversity in programmer's preference, so some programmers prefer to write detailed comments and/or to use longer named local variables. Therefore, the notion of ownership [16] of methods may affect the results in this paper. Since our data did not show large variances among products (see Tables III and IV), the diversity in the programmer's preference does not seem to have an impact on the results.

Since our data collection focused only on the occurrences of modifications (including bug fixes), we missed the size of modifications. This causes a concern that we might overestimate the contribution of many minor changes. However, the fact remains that those methods require some changes after their release, so our results do not become worthless.

IV. Conclusion

We have empirically examined the heuristics that name of local variables should be shorter, in terms of change-proneness in Java methods. As the results, we obtained the following two findings: a method having a longer named local variable is more change-prone, and the presence of comments inside the method body strengthens the suspicions to be modified after the release. Our future work includes: (1) a further analysis with varying combination of the variable name's length and the presence of inner comments, (2) analyses on other large scale products written in languages other than Java, (3) an analysis of the effects of ownership on change-proneness, (4) an empirical study on the impact of a variable's scope length and/or its type, and (5) a semantic analysis of variables' names.

Acknowledgment

This work was supported by JSPS KAKENHI #25330083.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Longman, 1999.

[2] H. Aman, "An empirical analysis on fault-proneness of well-commented modules," in *Proc. 4th Int'l Workshop Empir. Softw. Eng. in Practice*, Oct. 2012, pp. 3–9.

[3] ——, "An empirical analysis of the impact of comment statements on fault-proneness of small-size module," in *Proc. 19th Asia-Pacific Softw. Eng. Conf.*, Dec. 2012, pp. 362–367.

[4] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of comments and fault-proneness in methods: can comments point to faulty methods?" in *Proc. 8th Int'l Symp. Empir. Softw. Eng. and Measurement*, Sept. 2014, p. 63.

[5] ——, "Empirical analysis of fault-proneness in methods by focusing on their comment lines," in *Proc. 21st Asia-Pacific Softw. Eng. Conf.*, vol. 2, Dec. 2014, pp. 51–56.

[6] B. W. Kernighan and R. Pike, *The practice of programming*. Boston, MA: Addison-Wesley Longman, 1999.

[7] Free Software Foundation, "Gnu coding standards," https://www.gnu.org/prep/standards/.

[8] Sun Microsystems, "Code conventions for the java programming language," http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.

[9] M. J. Sousa and H. Moreira, "A survey on the software maintenance process," in *Proc. Int'l Conf. Softw. Maintenance*, Nov. 1998, pp. 265–274.

[10] T. Tenny, "Program readability: Procedures versus comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, Sept. 1988.

[11] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proc. 5th Int'l Conf. Softw. Eng.*, Mar. 1981, pp. 215–223.

[12] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. 21st Int'l Conf. Program Comprehension*, May 2013, pp. 83–92.

[13] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proc. 14th Int'l Conf. Program Comprehension*, June 2006, pp. 3–12.

[14] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int'l Workshop on Mining Softw. Repositories*, May 2005, pp. 1–5.

[15] A. Agresti, *Categorical Data Analysis*, 2nd ed. N.J.: Wiley, 2002.

[16] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proc. 35th Int'l Conf. Softw. Eng.*, May 2013, pp. 452–461.