Local Variables with Compound Names and Comments as Signs of Fault-Prone Java Methods

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

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

Minoru Kawahara
Center for Information Technology
Ehime University
Matsuyama, Ehime 790-8577, Japan

***Abstract*—This paper focuses on two types of artifacts—local variables and comments in a method (function). Both of them are usually used at the programer's discretion. Thus, naming local variables and commenting code can vary among individuals, and such an individual difference may cause a dispersion in quality. This paper conducts an empirical analysis on the fault-proneness of Java methods which are collected from nine popular open source products. The results report the following three findings: (1) Methods having local variables with compound names are more likely to be faulty than the others; (2) Methods having local variables with simple and short names are unlikely to be faulty, but their positive effects tend to be decayed as their scopes get wider; (3) The presence of comments within a method body can also be useful sign of fault-prone method.**

I. Introduction

Software systems have been utilized in many aspects of our daily life, and management of software quality has been the most significant activity for ensuring the safety and security of the people. In fact, it is hard to always make a one-shot release of a perfect software product which has no need to be enhanced or modified in the future; software systems usually require upgrades after their releases in order to fix their faults and/or to enrich their functionality. Needless to say, it is better to reduce both the frequency of their upgrades and the size of their patches to be applied.

To minimize upgrades of software products, thorough review and testing before their releases are desirable activities. In general, software review and testing help to detect concealed faults or identify suspicious software modules which are fault-prone [1], [2]. Then, those problems can be resolved by fixing faults or refactoring problematic programs in order to reduce the risk of causing unwanted upgrades after their releases. While review and testing are useful activities, they are also costly ones, so there have been many studies using software metrics to predict fault-prone modules prior to software review and testing activities [3]. By predicting fault-prone parts of a software product, cost-effective review and testing would be performed, i.e., we would detect more faults at less cost.

Most studied methods and models for predicting fault-prone modules have been based on structural features of products such as their sizes and complexities, or on development histories stored in their code repositories such as the number of bug-fix commitments which have been made by a certain point in time [4], [5], [6]. However, the impact of human factors would also be significant since programming activities are usually done by human beings. Different programmers would probably develop different programs for the same specification. Such a difference among individuals must have a certain level of influence on the quality of products, i.e., it must cause a dispersion in quality. Therefore, we focus on the following two artifacts which may vary from person to person, (1) local variables declared in a method (function) and (2) comments written inside the method body. While these artifacts have no impact on the structure of a program, they seem to be related to the understandability and the readability of the program, so they can be expected to play important roles in predicting fault-prone methods. In this paper, we quantitatively analyze the relationships of these artifacts with the fault-proneness.

The key contribution of this paper is to provide the following findings derived from the results of our empirical analysis with nine popular open source software (OSS) products:

- Local variables with descriptive compound names (for example, "`countOfSatisfactoryRecords`") can be signs that the methods are fault-prone.
- Methods having local variables with simple and short names (for example, "`c`" or "`cnt`") are unlikely to be faulty, but their positive effects tend to be decayed as their scopes get wider.
- Comments within a method body also seem to be related to the fault-proneness of the method.

The remainder of this paper is organized as follows. Section II describes two types of artifacts which may vary among programmers—(1) names of local variables and (2) comments written inside a method body—and their relationships with the quality of source programs, and gives our research questions in regard to impacts of those artifacts. Section III reports on an empirical analysis on our research questions using popular OSS products, and discusses the results. Section IV briefly describes related work. Finally, Section V presents the conclusion of this paper and our future work.

II. Local Variables and Comments

This paper focuses on local variables and comments, since they may vary widely from person to person and cause a variation in quality. This section describes concerns of local variable names and comments in regard to source code quality, and set up our research questions.

A. Local Variable Name

Since local variables are valid only within a function or a method, names of local variables are usually not specified in their software specifications or design documents. Therefore, naming local variables can be at the programmer's discretion. In general, different programmers would prefer different names for local variables even if they implement the same algorithm in their function or method. For example, a programmer likes to use "`count`" as the name of local variable for storing the number of records which satisfy a certain condition, but another programmer prefers "`c`" as its name; there might even be a programmer who wants to give "`countOfSatisfactoryRecords`" to the variable.

Needless to say, local variables with fully-spelled names such as "`count`" or ones with descriptive compound names "`countOfSatisfactoryRecords`" make it easy to understand the roles of those variables in their function or method since those names provide more information about those variables than shorter and/or simpler names. Lawrie et al. [7] surveyed the understandability of identifiers (including not only local variables' names but also functions' names) used in programs by comparing three types of names, (1) fully-spelled names such as "`count`," (2) abbreviated names such as "`cnt`" and (3) names using only an initial letter such as "`c`." They reported that a longer name is easier to understand for programmers, but there is not a significant difference in comprehensibility between fully-spelled names and abbreviated ones in their survey results. That is to say, it is not always necessary to give a long and descriptive name to a local variable, and a short and simple name may be sufficient.

There are also programming heuristics on naming local variables. Both the GNU coding standards [8] and the Java coding convention [9] have said that names of local variables should be shorter. Moreover, Kernighan and Pike [10] also argued that shorter names are sufficient for local variables; for example, they considered that name "`n`" looks good for a local variable storing "the number of points" while name "`numberOfPoints`" seems to be overdone. Thus, long and descriptive names have not been recommended for the names of local variables. However, the impact of such a descriptive name on the code quality has not been clearly discussed in those heuristics.

Aman et al. [11] conducted an empirical work and showed that methods having local variables with long names are more likely to be fault-prone and change-prone than the other methods. That is to say, they showed a relationship between a long name of a local variable and a poor quality of the code in a statistical manner. However, their analysis missed taking into account the following two aspects: (1) the composition of variable's name and (2) the scope of local variable. Focusing on not only the length of local variable's name but also those two aspects would be more worthy in analyzing the impact of local variable's name and in enhancing the quality of code. This is a key motivation of this work.

B. Comments

Comments are documents embedded in a source file, which usually provide beneficial information in regard to the program [12]. While there are several types of comments, we focus on comments written inside a method (function) body in this paper. Those comments usually give explanations or programmer's memos for their implementation in the method. Of course, the other types of comments also provide important information regarding the program. However, such comments written outside a method body are often the copyright designation or the programmer's manual explaining how to use the method, i.e., those comments may not be decided at the discretion of the programmer. Thus, those comments outside a method body may be out of our research scope focusing on the individual difference among programmers. That is the reason why we will focus only on the comments written inside a method body.

While comments along with executable code can be a great help in understanding the code, there have also been criticisms on their effects: comments might be written to compensate for a lack of readability in complicated programs [13]. In this context, Fowler [14] pointed out that well-written comments may be "deodorant" for masking "code smells." Although comments themselves are good artifacts, they may be used for neutralizing a "bad-smelling" code. Kernighan and Pike [10] said that programmers should not add detailed comments to a bad code; in such a case, it is better to rewrite their code rather than adding comments. If a programmer wants to add detailed comments to their code during their programming activity, the programmer may consider that the program is hard to understand for others without those comments. That is to say, comments may be signs of complicated programs. Aman et al. [11], [15] reported supporting empirical results that commented programs tend to be more fault-prone than non-commented ones. In this paper, we conduct a further analysis examining combinations of (1) the composition of local variable's name, (2) local variable's scope and (3) comments, in terms of fault-proneness.

C. Research Questions

As mentioned above, both the local variables and the comments are not only artifacts which may vary among programmers, but also remarkable ones which are expected to have relationships with the quality of the code. However, the analyses in the previous work [11], [15] missed considerations for the composition of local variable's name and the scope of local variable. We will conduct a further analysis by focusing on those missed aspects as well. In order to clarify our points of view in our empirical analysis, we set up the following two research questions (RQs):

TABLE I
SURVEYED OSS PRODUCTS.

Product	Size (KLOC)	#Methods Having a Local Variable	Data Collection Period	Domain
IP-Scanner	16	433	2006-07-19 — 2016-04-04	Networking
Checkstyle	21	738	2003-05-05 — 2016-03-28	Code analysis
eXo	21	675	2007-03-17 — 2016-04-06	Social collaboration software
FreeMind	71	$2,353$	2011-02-06 — 2016-03-30	Mind-mapping tool
ARM	282	$1,300$	2013-09-11 — 2016-03-14	Development support
Hibernate	387	$6,372$	2007-06-29 — 2016-03-31	Object/Relational mapping
ProjectLibre	224	$1,466$	2012-08-22 — 2016-04-06	MS Project clone
PMD	75	738	2002-06-21 — 2016-04-05	Source code analyzer
SQuirreL	405	$6,060$	2001-06-01 — 2016-04-05	Database client
Total	$1,502$	$20,135$		

RQ1 Can local variables with compound names be signs of fault-prone methods?

RQ2 How does a local variable's scope relate to the effect of local variable's name on the fault-proneness in a method?

We will check the above two questions while considering the impact of comments as well.

□

As mentioned in Section II-A, there have been concerns in giving descriptive names to local variables. Compound names such as "`numberOfPoints`" are typical descriptive names. RQ1 asks whether a local variable with such a compound name can be a sign to find fault-prone method or not.

If a local variable is declared with a narrow scope, it does not seem to need a descriptive name since its influence is limited within a narrow range. RQ2 focuses on the relationship of local variables' names with their scopes.

In examining these RQs, this paper expects to find yet another useful clue of fault-prone methods by focusing on their local variable names.

III. EMPIRICAL ANALYSIS

This section conducts an empirical analysis in which we collect quantitative data from popular OSS products and analyzes that data in order to discuss the above research questions.

A. Aim and Dataset

The aim of this analysis is to quantitatively examine the fault-proneness of Java methods by focusing on the names of local variables, the scopes of them and the presence of comments. The results of this analysis are expected to present useful points to be checked during code review activities.

We collected data from nine popular OSS products of different size and domain, shown in Table I—(1) Angry IP Scanner (IP-Scanner)[1], (2) Eclipse Checkstyle Plug-in (Checkstyle)[2], (3) eXo Platform (eXo)[3], (4) FreeMind[4], (5) GNU ARM Eclipse Plug-ins (ARM)[5], (6) Hibernate ORM (Hibernate)[6], (7) PMD[7], (8) ProjectLibre[8] and (9) SQuirreL SQL Client (SQuirreL)[9]. All of them are ranked in the top 50 popular Java products at SourceForge.net[10], and their source files have been maintained with the Git. The restrictions of the development language and the version control system are from our data collection tools[11].

B. Procedure of Data Collection

We collected data from each OSS project in the following procedure.

(1) Make a clone of the repository, and make the list of all methods included in the current version.

(2) Get the change history of each method:
We check the source lines which had been changed through each commitment on the repository, and decide which methods were modified at that time (see Fig.1). The decision is made by the following three steps.

(2a) Get both the older version and the newer version of the source file which had upgraded through the commitment.

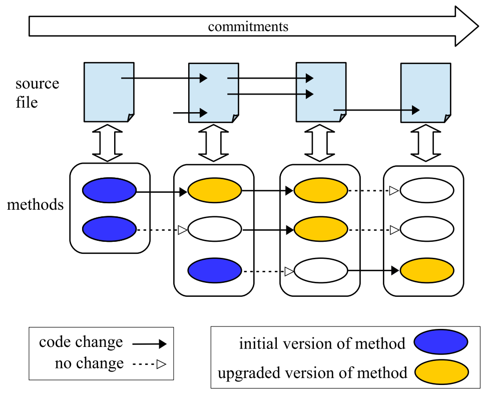


Fig. 1. Change histories of methods included in a source file.

[1]http://angryip.org/
[2]http://eclipse-cs.sourceforge.net/
[3]http://exoplatform.com/
[4]http://freemind.sourceforge.net/wiki/index.php/Main_Page
[5]http://gnuarmeclipse.livius.net/blog/
[6]http://hibernate.org/
[7]https://pmd.github.io/
[8]http://www.projectlibre.org/
[9]http://www.squirrelsql.org/
[10]http://sourceforge.net/
[11]http://se.cite.ehime-u.ac.jp/tool/

(2b) Compare those two consecutive versions, and find different parts between them. Then, obtain corresponding line numbers in the newer version.

(2c) Decide which method(s) had been upgraded, by checking the line numbers of upgraded lines against each method's position (range) in the newer version.

By iterating these steps for all commitments, we get the change history of each method.

(3) Collect the data on representative local variables' names and scopes, and comments for each method:
We survey names of local variables declared in the initial version of a method (see Fig.1) and the scopes of those variables. We define the length of a local variable's scope to be the number of lines where the variable is valid except for the line of its declaration. For example, the length of scope of variable "`len`" shown in Fig.2 is 5 and that of variable "`str`" is 2, respectively.
When there are two or more variables in a method, we focus on the variable whose scope is widest in the method as the "representative local variable" in order to connect the features of the local variable to the method. In the example shown in Fig.2, the "representative local variable" of method "`foo`" is variable "`len`." If there are two or more local variables with the widest scope in a method, we will adopt the variable with the longer name (having more characters) as the representative variable. Needless to say, if there is only one local variable in a method, the variable is the representative local variable of the method. On the other hand, any methods having no local variable are excluded from the data of interest in this work.
We collect the lines of comments written inside a method body as well.

(4) Check if a bug fix has occurred for each method:
We examine the change history of each method obtained above and check if a bug fix has occurred or not at the method's upgrade. We decide whether a code change was intended to a bug fixing or not, by checking their commitment message [16]. For example, Fig. 3 shows a part of commitment message (obtained by using `git log` command) on the repository of SquirreL SQL Client, which seems to be a bug fixing commitment. Since method "`_init`" in "`AliasEditController.java`" was modified through the commitment, we consider that a bug fixing was performed at the method.

C. Procedure of Data Analysis

We conducted our data analysis in the following procedure.

```
String foo (String arg) {
   int len = arg.length();
   if (len < 5) {
     return new String(arg);
   }
   String str = arg.substring(0, 5);
   return str + "...";
}
```

Fig. 2. An example of method having local variables.

```
commit 0d005dc6573dcc12df03917ee974a0736b4d5cfd
............
Bug #1236 Shortcut for comment/uncomment current line
(ctrl + "/") does not
Fixed according to the suggestion in the bug #1236
Please note: The orginal comment/uncomment hot key of
SQuirreL is ctrl+Num
```

Fig. 3. An example of actual commitment message.

(1) Perform a random sampling of methods, which have a local variable, from all projects:
In order to avoid an impact of project's size bias on our empirical results, we randomly sample the same number of methods from each project.

(2) Divide the set of methods into subsets according to the representative local variable's name.
We consider "a local variable with a short name" to be one such that the length of its name is less than or equal to the 25 percentile in the distribution of length of name. We also take into account if the name is compound one or not for RQ1. Thus, we consider the following three categories.
- V_1: the set of methods such that the name of representative local variable is *short* and *not compound*.
- V_2: the set of methods such that the name of representative local variable is *not short* and *not compound*.
- V_3: the set of methods such that the name of representative local variable is a *compound* one.

We decide that a variable has a compound name if it is composed in camel case such as "`numberOfItems`." That is to say, we consider a name to be compound one if it has a lower case letter followed by an upper case letter. We regard such a pair of lower case letter and upper case letter as a splitting position of the name. For example, there are two splitting positions in "`numberOfItems`," i.e., the pair of "`r`" and "`O`," and the pair of "`f`" and "`I`," so the name can be split into three portions (words) "`number`," "`Of`" and "`Items`." We consider that such compounded names cannot be short ones composed by at most a few characters. Thus, we do not divide the set of methods having representative local variables with compound names, and define V_3 only (not V_3 and V_4).

(3) Divide the subsets of methods obtained at Step (2) into two, according to the presence of comments:
In order to analyze the impact of comments as well, we divide the set of methods into two subsets by checking if there are comments[12] inside method bodies or not.
- C_0: the set of methods having *no comment*.
- C_1: the set of methods having *comments*.

Then, we define $M_{ij} = C_i \cap V_j$ for $i = 0, 1$ and $j = 1, 2, 3$. For example, M_{01} is the set of non-commented methods in which the representative local variable has a short and non-compound name. Table II summarizes these categories (the method sets) M_{ij}.

[12]We excluded the comment out cases from our data by using a checking algorithm [17].

TABLE II
SYMBOLS REPRESENTING CATEGORIES.

Symbol	Name of representative local variable		
	Non-compound		Compound
	short	not short	
Non-commented methods	M_{01}	M_{02}	M_{03}
Commented methods	M_{11}	M_{12}	M_{13}

(4) Examine the fault-proneness of methods by the above categories M_{ij}:
We statistically compare the bug fix rates among categories M_{ij} (for $i = 0, 1$ and $j = 1, 2, 3$) and discuss the results.

(5) Examine the trends of the bug fix rates over scope:
In order to analyze the impact of variable's scope as well, we analyze the changes in bug fix rate by varying the range (the length of scope) which we focus on. In the concrete, we compare the moving averages of the bug fix rates among categories M_{ij} (for $i = 0, 1$ and $j = 1, 2, 3$), by varying the range of focusing scope.

D. Results and Discussion: Collected Data

We first show the results of our data collection. Since the minimum number of methods included in a project was 433 as shown in Table I (project "IP-Scanner"), we randomly sampled 400 methods from each project, so our dataset consists of $3,600$ methods in total.

Table III shows the distributions of length of representative local variables' names in character count and in word count, respectively. Here, "word count" means the number of words composing a variable's name which is split according to the notion of the camel case. The longest names in character count were "`containsSuppressWarningsHolderModule`" and "`organizationInitializersHomePathNode`" which consist of 36 characters, and the longest name in word count was "`thereWereNodesToBeFolded`" which consists of 6 words. Although such some long and descriptive names appear in some methods, most local variables have names that consist of at most a few characters and they are non-compound names whose word count is one. Since the 25 percentile (Q_1) of the character count is four as shown in Table III, we will consider a name whose length is less than or equal to four letters to be short in the following analysis.

Table IV presents the distribution of length of a representative local variable's scope. Since there were some methods as shown in Fig.4, where the minimum length of the scope is zero. As all local variables are valid only within a (part of the) method, the majority of them are around a few to ten lines of code. In order to filter out extreme data which may be noise in our analysis, we will use only the data whose scopes are in between 25 percentile ($Q_1 = 4$) and 75 percentile ($Q_3 = 19$) of their distribution. By this data filtering, the number of our samples are reduced to $1,872$. Table V gives the number of methods belong to each category M_{ij} (for $i = 0, 1$; $j = 1, 2, 3$) after this filtering.

TABLE III
DISTRIBUTION OF LENGTH OF REPRESENTATIVE LOCAL VARIABLE NAMES.

Unit	Min.	Q_1	Median	Q_3	Max.
Character	1	4	6	10	36
Word	1	1	1	2	6

(Q_1: 25 percentile; Q_3: 75 percentile)

TABLE IV
DISTRIBUTION OF SCOPE OF REPRESENTATIVE LOCAL VARIABLES.

Min.	Q_1	Median	Q_3	Max.
0	4	9	19	793

(Q_1: 25 percentile; Q_3: 75 percentile)

```
private void doConnectToRunningChanged() {
  if (doStartGdbServer.getSelection()) {
    boolean enabled = doConnectToRunning.getSelection();
  }
}
```

Fig. 4. An instance of local variable whose scope is zero ("enabled").

Table VI shows the distribution of the number of bug fixes which had occurred in methods over their upgrades. About 18% of methods seemed to have had a hidden fault and have fixed through their code changes. Since we already filtered out the methods such that the scope of the representative local variable was wide, most of the methods in our dataset were small-sized and thus possibly more simple in structure. Hence, conventional size metrics and structural complexity metrics would be ineffective for analyzing the fault-proneness of methods in detail. It would be worth it to focus on a feature of methods other than the size and complexity. A local variable name might be yet another useful feature to be focused on.

E. Results and Discussion: Comparison of Bug Fix Rates by Category

Table VII presents the bug fix rate in each category M_{ij} (for $i = 0, 1$; $j = 1, 2, 3$). There seem to be differences in the

TABLE V
NUMBER OF METHODS BELONG TO EACH CATEGORY.

Category	Non-Compound		Compound Name	Total
	≤ 4	> 4		
Non-Commented	401 (M_{01})	527 (M_{02})	427 (M_{03})	$1,355$ (C_0)
Commented	139 (M_{11})	164 (M_{12})	214 (M_{13})	517 (C_1)
Total	540 (V_1)	691 (V_2)	641 (V_3)	$1,872$

TABLE VI
DISTRIBUTION OF NUMBER OF BUG FIXES OBSERVED IN METHODS AND BUG FIX RATE.

Min.	Q_1	Median	Q_3	Max.	Rate
0	0	0	0	5	18.1%

(Q_1: 25 percentile; Q_3: 75 percentile) bug fix rates among categories. The minimum bug fix rate is 0.135 in M_{01} and the maximum bug fix rate is 0.252 in M_{13}, so the latter rate is about twice larger than the former one.

We did a χ^2 test for the differences of bug fix rates in the results. The test confirmed that there are statistically significant differences among the bug fix rates in the categories, at $p = 0.0053 < 1\%$ level of significance ($\chi^2 = 16.6$; degree of freedom $= 5$). That is to say, the above categorization of methods by focusing on the name of local variables and comments is meaningful for discussing the differences of fault-proneness in the methods.

In the categories of non-commented methods M_{0j} (for $j = 1, 2, 3$), we can observe an increasing trend in the bug fix rate (BFR): $\mathrm{BFR}(M_{01}) = 0.135 < \mathrm{BFR}(M_{02}) = 0.165 < \mathrm{BFR}(M_{03}) = 0.211$ (see Table VII and Fig.5(a)). We also identified that the increasing tendency is statistically significant through the Cochran-Armitage test [18] at $p = 0.0035 < 1\%$ level of significance ($\chi^2 = 8.52$; degree of freedom $= 1$). From this trend, we can say that methods having representative local variables with shorter names are likely to be better in terms of fault-proneness, and the ones with compound names are worse than others.

On the other hand, in the categories of commented methods M_{1j} (for $j = 1, 2, 3$), we cannot identify an increasing trend in the bug fix rate; they seems that $\mathrm{BFR}(M_{11}) = 0.180 \simeq \mathrm{BFR}(M_{12}) = 0.177 < \mathrm{BFR}(M_{13}) = 0.252$ (see Table VII and Fig.5(b)).

For all three categories, their bug fix rates were higher than ones of non-commented methods, i.e., $\mathrm{BFR}(M_{0j}) < \mathrm{BFR}(M_{1j})$ (for $j = 1, 2, 3$):

- $\mathrm{BFR}(M_{01}) = 0.135 < \mathrm{BFR}(M_{11}) = 0.180$,
- $\mathrm{BFR}(M_{02}) = 0.165 < \mathrm{BFR}(M_{12}) = 0.177$, and
- $\mathrm{BFR}(M_{03}) = 0.211 < \mathrm{BFR}(M_{13}) = 0.252$.

Thus, the commented methods seem to be riskier in fault-proneness than the non-commented methods. Similar trends in regard to comments have been reported in the previous work [11], [15] as well. Since programmers might want to add comments when they considered that their code is difficult to understand without an explanation, the presence of comments would be a sign indicating that the code is complicated.

Notice that the bug fix rates in the categories of compound names, M_{03} and M_{13}, are the highest ones among categories; Only those two categories show bug fix rates which are higher than the average of all (18.1%) (see Fig.5). Thus, the methods having representative local variables with compound names are likely to be fault-prone regardless of the presence of comments.

TABLE VII
BUG FIX RATES BY CATEGORY.

Category	Non-Compound name ≤ 4		Non-Compound name > 4		Compound name	
Non commented	M_{01}	0.135 $(\frac{54}{401})$	M_{02}	0.165 $(\frac{87}{527})$	M_{03}	0.211 $(\frac{90}{427})$
Commented	M_{11}	0.180 $(\frac{25}{139})$	M_{12}	0.177 $(\frac{29}{164})$	M_{13}	0.252 $(\frac{54}{214})$

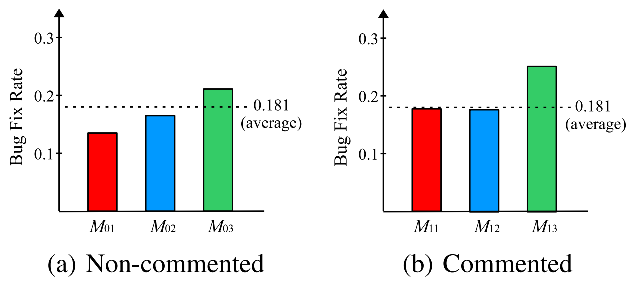


Fig. 5. Comparison of bug fix rates by category.

F. Results and Discussion: Comparison of Bug Fix Rates over Scope

This subsection compares the bug fix rates among the categories from another in-depth perspective of local variable's property, "scope."

We first checked correlations of the length of a local variable name with its scope. There do not seem to be specific correlation between the length of local variable's name and the length of its scope (see Fig.6): Spearman rank-correlation coefficients in character count and in word count were 0.083 and 0.0003, respectively. Hence, the length of a local variable's name is statistically independent of the length of its scope, and the scope is not a confounding factor for discussing the fault-proneness of methods by using their local variable's name.

To observe the changes in fault-proneness over variable's scope, we computed the moving averages of bug fix rates by varying the focusing interval of scope $[s-5, s+5]$ for $s = 9, 10, \ldots, 14$; in simplified terms, we obtained the bug fix rates of methods whose representative local variable's scope is "around s" ($s \pm 5$), where the lower and the upper limit of s are decided so as to keep the interval $[s-5, s+5]$ within the scope range of all data: between 4 and 19. For example, if $s = 9$ then $[4, 14]$ is the focusing interval, we focus only on the methods whose representative local variable's scope is "around 9" (9 ± 5). Figure 7 shows those results.

In Fig.7(a), we observed the relationships of bug fix rates regardless of scope: $\mathrm{BFR}(M_{01}) < \mathrm{BFR}(M_{02}) < \mathrm{BFR}(M_{03})$,

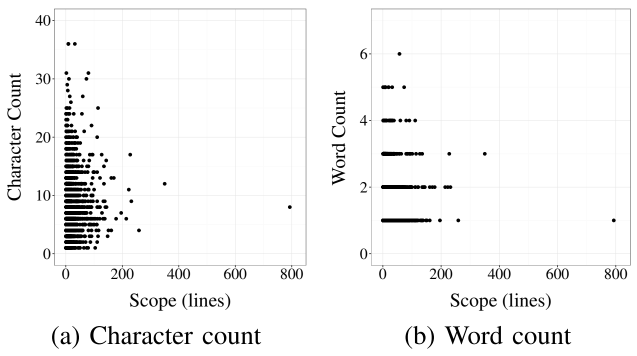


Fig. 6. Scatter diagrams: the length of variable's name vs. the length of variable's scope.

which are similar to the results shown in Fig.5(a). Thus, we can say with emphasis: while the fault-proneness of methods having representative local variables with shorter names are low, the methods having representative local variables with compound names are high. Since the gap in the bug fix rate between M_{01} and M_{02} becomes smaller as the scope gets wider, the superiority of a shorter name may be limited to a narrower scope. If a local variable with a short and simple name is used in a wider scope, it might cause an abuse of the variable or a poor understandability of the program's behavior. While Kernighan and Pike [10] said to give a short and simple name to a local variable, they did not recommend such a naming in any case, and their argument supposed the case that a local variables was used in just "locally" within a part of a program. The results observed in Fig.7(a) seem to support such a programming heuristic.

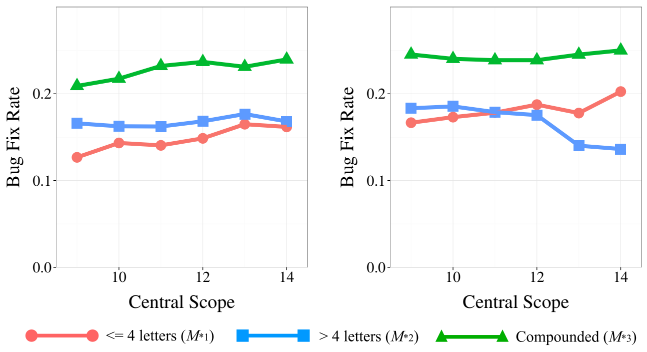
In Fig.7(b), while M_{11} ($\leq$ 4 letters) are better than M_{12} ($>$ 4 letters) with narrower scopes around 9 or 10, their magnitude relationship inverts as their scope gets wider. That would be the reason why $\mathrm{BFR}(M_{11}) \simeq \mathrm{BFR}(M_{12})$ in Fig.5(b). Therefore, we can say that a shorter name is better with a narrower scope, but cannot claim a shorter name is *always* better. If programmers wanted to add comments, there would be a lack of clarity in their code. In such a case, a shorter name with a wider scope might spur the program's poor comprehensibility. On the other hand, compound names always show the worst (highest) bug fix rates regardless of scope, similar to the results in Fig.7(a). Although compound names are usually descriptive, they seem to be signs of fault-prone methods. If a programmer wanted to give a compound name to a local variable, the role of the variable would be somewhat complicated, so methods having such local variables might be riskier than the others in terms of fault-proneness.

G. Answers to RQs

From the results of Sections III-E and III-F, we summarize our findings for RQ1 and RQ2 in the following.

For RQ1, we conclude that methods having local variables with compound names are likely to be faulty regardless of scope. Although we do not imply that compound names cause faults in methods, the presence of a local variable with a compound name may be a clue finding a risky part from the perspective of the fault-proneness in a method. Such a local variable might have an important role or a more complex role in the program, so they have to be reviewed more carefully.

For RQ2, we can say that shorter names are better for local variables with narrower scope. As a scope gets wider, the positive effect of shorter names seems to be decayed. While a short and simple name would be preferable as mentioned in some coding conventions and programmers' heuristics [8], [9], [10], our empirical results quantitatively showed that the variable's scope is also a feature worthy of consideration. Moreover, the presence of comments may degrade the superiority of shorter names as their scopes get wider. Therefore, we should take into account not only the composition of local variable's name but also its scope and comments in the code review.



(a) Non-commented (b) Commented

Fig. 7. Moving averages of bug fix rates over scope.

H. Threats to Validity

This empirical analysis has been conducted for Java products. While another programming language might produce different results, there would not be essential differences in the concept of local variables and comments, among Java and many other modern programming languages. Thus, the difference in programming language would not be a serious threat to validity.

In order to avoid the data selection bias, we adopted a random sampling in our data collection. Moreover, we used popular different sized OSS products from different domains. Therefore, our construction of dataset would not be a threat to validity.

Since our data is collected from the initial version of the methods, some methods might be no longer used today. However, all methods in our dataset are included in the latest version of the product because we made our method list by checking the latest version of their source files as described in Section III-B. Moreover, we did a random sampling from them. Thus, we consider it will not be a serious threat to validity in our empirical work.

Our definition of compound name is based on the notion of camel case. If there are local variables whose names are composed by another rule such as the snake case, e.g., "`number_of_items`," they are wrongly categorized into non-compound names. Thus, we rechecked all representative local variables' names included in our data set, then we found only two variables having snake case names, "`s_descriptors`" and "`size_h`." Due to the small number of error cases, our name splitting method was not a serious threat to validity.

IV. Related Work

Lawrie et al. conducted a survey on names of identifiers in terms of their comprehensibility for over 100 programmers [7]. In their survey research, they classified names of identifiers into three categories (1) fully-spelled name, (2) abbreviated name and (3) initial letter—for example, (1) "`count`," (2) "`cnt`" and (3) "`c`"—, then compared their understandability.

Their results showed that fully-spelled names were the easiest to understand but that there did not seem to be significant differences with abbreviated names in their comprehensibility level. While their work provides a useful motivation to study whether a shorter name is better or not, they did not discuss the fault-proneness of program.

Kawamoto and Mizuno [19] conducted an empirical study with two OSS products and reported that a class including a long identifier tends to be faulty. While their work is one of our most significant previous studies, our work focuses on a finer-grained artifact—local variable—and conducts a statistical analysis with taking into account of the scopes and the comments.

Binkley et al. [20] focused on the relationship between the length of identifier (including a variable's name, a method's name and a class's name) and the human short-term memory. They identified that identifiers with long names are related to a difficulty in program comprehension. They were concerned that a long chain, e.g., "`class.firstAssignment().name.trim()`," would cause a loss of the readability of the code. While the research viewpoint differs from our work, the fundamental concern about the length of name is common, and it seems to be well accorded with our results showing the compound names are not recommended for local variables.

Aman et al. [11] reported an empirical analysis showing that Java methods having local variables with long names are more likely to be fault-prone and change-prone than the other methods. That report is our significant previous work, and this paper focuses more detailed features of local variables, i.e., the composition of name and their scopes. While another work by Aman et al. [15], reporting that commented programs tend to be more fault-prone, is also our important previous work, we conduct a further analysis examining combinations of the local variable's name, the scope and the comments in this paper.

V. Conclusion

We have focused on programming artifacts which may vary among individuals: local variables' names and comments. Popular code conventions say that names of local variables should be shorter and simple, and it seems to have been a heuristic of programmers. We empirically evaluated the heuristic in terms of fault-proneness by checking the names of local variables, their scopes and the presence of comments. The empirical analysis for the data from nine popular OSS products showed the following three findings.

(1) Local variables with compound names can be signs of fault-prone methods.
(2) Methods having the representative local variables with non-compound and shorter names (≤ 4 letters) are less fault-prone, but their positive effects are decayed as their scopes get wider (around 10 or more lines).
(3) Methods having comments in their bodies are also more likely to be faulty.

These findings are expected to be useful guidelines for more efficient code reviews.

One of our significant future works is to conduct further analyses of local variables' names, which include an application of the natural language processing technologies to evaluate the meaning of local variables' names. A further analysis with products written in a programming language other than Java is also our future project in order to ensure the generality of the above findings.

Acknowledgment

This work was supported by JSPS KAKENHI #16K00099. The authors would like to thank anonymous reviewers for their helpful comments.

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. N.J.: John Wiley & Sons, 2004.
[2] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proc. 9th Joint Meeting on Foundations of Softw. Eng.*, Aug. 2013, pp. 202–212.
[3] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc." in *Proc. 28th Int'l Conf. Softw. Eng.*, May 2006, pp. 413–422.
[4] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, July 2008.
[5] Y. Liu, T. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 852–864, Nov 2010.
[6] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. 2013 Int'l Conf. Softw. Eng.*, May 2013, pp. 432–441.
[7] 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.
[8] Free Software Foundation, "Gnu coding standards," https://www.gnu.org/prep/standards/.
[9] Sun Microsystems, "Code conventions for the java programming language," http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.
[10] B. W. Kernighan and R. Pike, *The practice of programming*. Boston, MA: Addison-Wesley Longman, 1999.
[11] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of change-proneness in methods having local variables with long names and comments," in *Proc. 2015 ACM/IEEE Int'l Symp. Empirical Softw. Eng. and Measurement*, Oct. 2015, pp. 50–53.
[12] 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.
[13] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proc. 2008 Int'l Symp. Softw. Testing and Analysis*, 2008, pp. 121–130.
[14] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Longman, 1999.
[15] 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.
[16] 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.
[17] H. Aman, "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.
[18] A. Agresti, *Categorical Data Analysis*, 2nd ed. N.J.: Wiley, 2002.
[19] K. Kawamoto and O. Mizuno, "Predicting fault-prone modules using the length of identifiers," in *Proc. 4th Int'l Workshop on Empirical Softw. Eng. in Practice*, Oct. 2012, pp. 30–34, Japan.
[20] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," *Science of Computer Programming*, vol. 74, no. 7, pp. 430–445, May 2009.