Empirical Study of Abnormalities in Local Variables of Change-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*—The naming of local variables is usually at the programmer's discretion. Thus, there is a diversity in naming local variables and this may cause variations in the code quality. Many coding conventions say that the name of a local variable can/should be short. This paper focuses on such conventions, and aims to explore the trends of local variables' names in Java and examine if abnormal local variables create harmful effects on the code quality. This paper collected data on local variables—names, scopes and types—from six popular open source products, and proposes to evaluate their abnormalities by using the notion of the Mahalanobis distance. The empirical results report the following findings: 1) The trend of naming local variables differs according to the variable type; 2) The majority of local variables have short names with narrow scopes, where a name is often a word or its abbreviated form; 3) Methods having abnormal local variables are about 1.2 – 2.5 times more likely to be change-prone than the others. While the naming of local variables depends on who writes the code, there seem to be common trend of naming. Java methods with deviant local variables tend to be fixed many times after their release and cannot survive unscathed.**

I. Introduction

Programmers use variables to store values or refer to objects in their programs. In most procedural or object-oriented languages, variables are classified into two groups: global variables and local variables. While global variables are available everywhere in a program, local ones can be used only within a method (function) or a narrower block. Since the manipulation of a global variable has a risk of causing side effects in other parts of the program, it is better to use local variables instead of global ones to prevent fault creation [1].

Programmers can freely decide the names of local variables in many cases because of their localized scopes. Thus, the naming of local variables can vary considerably from person to person. Such a diversity in naming local variables may cause a variation in the code readability and understandability. Many coding conventions and practices give common recommendations in regard to the naming of local variables: names of local variables can/should be short (e.g., [1], [2], [3]). Needless to say, a longer and more descriptive name for a local variable would make it easier to understand its role. However, Lawrie et al. [4] empirically showed that the comprehensibility of code does not significantly differ between a fully-spelled name and its abbreviated form (e.g., `index` vs. `idx`). Binkley et al. [5] focused on the human short-term memory and showed that a long identifier name degrades the readability of code. Kernighan et al. [1] said that it is overdone to give a long and descriptive name to a local variable with a narrow scope. Based on these various reports and recommendations, we have the following simple questions which are our motivations: "What are the real trends of local variables' names and scopes?" and "Is an abnormal variable harmful to the quality of code?."

In this paper, we conduct an empirical analysis on local variables collected from open source software (OSS) products written in Java, and report the trends of local variables and the impact of abnormal variables on the quality of code from the perspective of the code change-proneness. The key contributions of this paper are as follows: 1) reports of empirical data in regard to local variables' names, scopes and types; 2) an application of the concept of the Mahalanobis distance to the abnormality evaluation of local variables; 3) reports on the relationship between the presence of an abnormal variable in a Java method and the method's change-proneness.

The remainder of this paper is organized as follows: Section II describes properties of local variables. Section III presents preferred features of local variables and our research questions (RQs), then defines the abnormality of local variables to tackle our RQs in a quantitative way. Section IV reports the results of data analyses on six popular OSS products, and gives discussions on them. Section V briefly describes related work. Finally, Section VI presents our conclusions and future work.

II. Properties of Local Variables

In preparation for our analysis of local variables, we briefly describe major properties of local variables in this section. Notice that our descriptions are aimed at Java programs.

A. Name

Any local variable has its own name. Giving a name to a local variable is almost always at the programmer's discretion. That is to say, requirement specifications and design documents usually do not specify names of local variables, so programmers can give any names to their local variables

during their programming activities. Hence, the naming of local variables is affected by the programmer's preference or custom in many cases. For a local variable, different programmers may want to give different names. The names of local variables are the major properties which can vary according to the programmer. The names of local variables are classified into the following two categories: 1) A single alphabetical character; 2) An alphabetical character followed by one or more alphanumeric character(s), where the underscore character ("_") is also included in the set of alphabetical characters.

A name according to category 1) is often the initial character of the word which expresses the role. Category 2) can be divided into subcategories: 2a) A fully-spelled word; 2b) An abbreviated word; 2c) A composition of two or more words; 2d) Another string.

For example of an index of an array, "`i`" is often used for a variable in category 1); names corresponding to 2a) and 2b) can be "`index`" and "`idx`," respectively. For 2c), compound names may be "`indexOfArray`." In Java, a compound name is usually made in the style of camel case. In camel case, each word or abbreviation begins with a capital letter except for the first word (abbreviation), e.g., "`indexOfArray`." Although we can also give a name in which all words (abbreviations) begin with capital letters like "`IndexOfArray`," it is a common practice to begin a name of local variable with a lower-case character; the former camel case and the latter one are called "lower camel case" and "upper camel case," respectively. In snake case, words or abbreviations are concatenated with the underscore character, e.g., "`index_of_array`." The (lower) camel case is the most popular style in Java; the snake case tends to be widely used in C/C++.

A name according to 2a), 2b) or 2c) is more descriptive than the ones of category 1). Thus, such names of local variables tend to make it easier to understand the role [4]. On the other hand, a name corresponding to 2d) might be hard to understand by others—the author is possibly the only person who can understand it. That might require comments or an external document.

B. Scope

A local variable is valid within a certain range of source code lines, which is referred to as its "scope." When a local variable is declared in a block, the variable's scope ranges from its declaration line to the end of the block.

In general, a narrower scope is better for a local variable. A local variable with a wider scope is likely to be involved in more executable statements, and it would make more complex data dependencies. Consequently, local variables having wide scopes may cause the poor readability of program which is one of the major features observed in fault-prone programs.

A local variable's scope can relate to the variable's name. When a variable has a narrow scope, there is no need to carefully explain the role of the variable because of its limited range. That is to say, we do not have to give a descriptive name to the variable in order to express what its task is. For example, "`i`" would be acceptable as the name of the local variable which is the index of the array in a small "for" loop; while "`indexOfArray`" makes it easier to understand its role, it may be overdone [1]. If the loop has a long and complicated block, it might be better to give a more descriptive name.

C. Type

Types of local variables in Java are categorized into two groups: 1) primitive types and 2) reference types [6]. The primitive types are fundamental data types predefined in the Java language specification: Boolean value, integer and floating point number. The reference types include the class types, the interface types and the array types: a class type variable is a reference to an instance of the class; an interface type variable is a reference to an object which has the interface; an array variable is a reference to an array of variables.

The type of a local variable may have an affect on the naming of the variable. For example, class types `Connection` and `Statement` usually appear in database related programs, and names of corresponding local variables are often `con` and `stmt`, respectively. When a local variable is an array, some programmers give a name in a plural form (e.g., `values`) or a name emphasizing that it is an array (e.g., `valueArray`).

III. Abnormality of Local Variables

A. Preferred Features and Research Questions

From the perspective of coding standards/conventions and programming practices, the following two principles have been preferred for local variables.

1) Short name: Kernighan and Pike [1], the Sun Java code convention [2] and the GNU coding standard [3] say that local variable names can/should be short.
2) Narrow scope: The GNU coding standard [3] and the Google Java style [7] say that local variables should be declared at the points which minimize their scopes.

Boswell et al. [8] also proposed giving a short name to a local variable if its scope is narrow, and a longer and more descriptive name to it if its scope is wide. Their proposal seems to be consistent with the above two principles.

As mentioned above, we have questions about the real trends of local variables and the relationship between the abnormality of local variables in a Java method and the method's change-proneness. Thus, we tackle the following RQs in this paper:

RQ1: What are the real trends of local variables in terms of their properties? (What are abnormal ones?)
RQ2: Is the presence of an abnormal local variable in a Java method related to the method's change-proneness?

RQ1 is a basic question. By collecting a lot of local variables and analyzing their properties, we will be able to see the de-facto standard for local variables and the relationships with the preferred features of local variables. Needless to say, it is natural that there are diversities in the properties of local variables. While considering such diversities, we will explore abnormal local variables in the real data and report the results.

RQ2 is an application of our abnormality evaluation for local variables: how can our empirical study contribute to the software engineering community? If the presence of an abnormal local variable in a method is harmful to the quality of code, the method cannot survive unscathed after its release. We will focus on the change-proneness of methods as our quality criterion in this paper.

B. Degree of Abnormality

In order to perform our data analysis on the above RQs, we have to quantify the abnormality of a local variable. We define the abnormality of local variables in the rest of this section:

1) Classify local variables into several categories according to their types (Sect. III-B1).
2) For each local variable in each category, represent the variable as the feature vector ($\boldsymbol{x}$) whose elements correspond to the variable's name and scope (Sect. III-B2).
3) For each category, compute the mean vector of feature vectors ($\boldsymbol{\mu}$) and the variance-covariance matrix (S), then use the Mahalanobis distance between $\boldsymbol{x}$ and $\boldsymbol{\mu}$ as the abnormality of a local variable corresponding to $\boldsymbol{x}$ (Sect. III-B3).

1) Classification of Local Variables: Types of local variables may be related to the naming of those variables. In order to avoid an impact caused by the difference in variable types, we will classify local variables into categories according to their types. In this paper, we distinguish between a primitive type and a reference type because a reference type variable can have richer information than a primitive type one. Moreover, we discriminate between an array and a single variable since an array is a set of variables[1]. Now we will exclude the reference types used for handling exceptions (or errors)[2] from our dataset since those exception variables are usually used only in exception handling blocks and often automatically generated by an integrated development environment such as Eclipse. Hence, we consider the following four categories of type: (i) primitive type, (ii) primitive array, (iii) reference type and (iv) reference array.

2) Feature Vector of a Local Variable: For each local variable in each category mentioned above, we consider a column vector $\boldsymbol{x} = (x_1, x_2, x_3)^T$, where x_1, x_2 and x_3 are the number of characters comprising the variable's name, the number of words comprising the name, and the number of lines where the variable is available, respectively[3]. We will call $\boldsymbol{x}$ the "feature vector" of the local variable in this paper. The first element—x_1: the number of characters comprising the local variable's name—represents the length of the name in character count. A long name possibly has an affect on the readability of the code [5]. The second element—x_2: the number of words comprising the name—is related to the amount of information which the name presents. A name with more words tends to be more descriptive. In this paper, we count the number of words in accordance with the camel case or the snake case. The third element—x_3: the number of lines in which the variable is valid—gives the length of scope. Popular coding conventions say that the scope of a local variable should be minimized [3], [7]. Moreover, the scope of a local variable may be related to the variable's name: for example, a programmer may dare to give a long and descriptive name to a local variable which has a wide scope.

3) Mahalanobis Distance: Let $\boldsymbol{\mu} = (\mu_1, \mu_2, \mu_3)^T$ be the mean vector of feature vectors in a type category. Given the feature vector $\boldsymbol{x}$ of a local variable, the distance between $\boldsymbol{x}$ and $\boldsymbol{\mu}$ can be the degree of abnormality which the variable has. In many cases, we see the Euclidean distance as a distance between two vectors (points), which is computed with:

$$|\boldsymbol{x} - \boldsymbol{\mu}| = \sqrt{(\boldsymbol{x} - \boldsymbol{\mu})^T (\boldsymbol{x} - \boldsymbol{\mu})} \ .$$

However, the Euclidean distance does not consider the dispersion of data and the correlations between their elements. Hence, we will use the Mahalanobis distance instead of the Euclidean distance. The Mahalanobis distance is a useful notion of distance to detect abnormal data while considering the distribution of data [9]. The rest of this subsection presents a brief explanation of the Mahalanobis distance.

The Mahalanobis distance between $\boldsymbol{x}$ and $\boldsymbol{\mu}$, $d(\boldsymbol{x}, \boldsymbol{\mu})$, is computed with the following equation:

$$d(\boldsymbol{x}, \boldsymbol{\mu}) = \sqrt{(\boldsymbol{x} - \boldsymbol{\mu})^T S^{-1} (\boldsymbol{x} - \boldsymbol{\mu})} \ , \qquad (1)$$

where $S = (\ \sigma_{ij}\ )$ is the variance-covariance matrix (for $i, j = 1, 2, 3$) and S^{-1} is its inverse matrix; σ_{ii} is the variance of the i-th element, and σ_{ij} is the covariance between the i-th element and the j-th one (for $i, j = 1, 2, 3$).

To give an intuitive interpretation of the Mahalanobis distance, let us consider the case of a one-dimensional vector, i.e., the case of scalar: $\boldsymbol{x} = x$ and $\boldsymbol{\mu} = \mu$. In this case, the Mahalanobis distance is computed with:

$$d(x, \mu) = \sqrt{(x - \mu) \frac{1}{\sigma^2} (x - \mu)} = \frac{|x - \mu|}{\sigma} \ . \qquad (2)$$

That is to say, the Mahalanobis distance is the Euclidean distance ($|x - \mu|$) normalized by the standard deviation (σ) of data. Equation (1) is the generalized form for a multi-dimensional vector.

By using the Mahalanobis distance, we can evaluate the abnormality of our object while considering their data distribution and correlations.

IV. Empirical Study

A. Aim and Dataset

The aims of this study are to report the results of empirical analyses in regard to local variables and to discuss the results.

[1] According to the Java language specification, an array is also of a reference type. However, we will not include arrays in our definition of reference type since we distinguish between an array and a single variable.

[2] We regard a class or interface type whose ancestor is `Exception`, `Error` or `Throwable` as an exception type.

[3] Symbol T signifies the transpose of a vector.

TABLE I
SURVEYED OSS PRODUCTS.

product	size (KLOC)	domain
Elasticsearch	635	search engine
Fastjson	124	JSON parser/generator
Guava	253	set of common libraries
libGDX	645	game-development application framework
Presto	431	distributed SQL query engine
RxJava	230	reactive extensions for JVM
total	2,048	

We explore six OSS products[4] shown in Table I. The key reasons why we used them are:

1) Their source files have been maintained with Git.
   Since this study requires a fine-grained source code analysis which explores local variables and code changes in methods, Git is suitable for our empirical study: Git allows us to make a clone of the repository on our local disk and provides powerful functionalities for analyzing source code and their changes.
2) Their source programs are written in Java.
   This reason comes from the limitations of the data collection tools we developed: JavaMethodExtractor and JavaVariableScopeExtractor[5].
3) They are popular and large-scale products.
   In general, it is hard to assure a generality of empirical results if they are obtained from minor or small-scale products, and those results would be less attractive for practitioners and researchers. Hence, we collected data from large-scale products that are as popular as possible: each of the above six products has source programs whose total size is larger than 100 KLOC, and is ranked in the top 30 products which have more "stars" at GitHub.

B. Procedures

We collect data of each product in the following procedure:

1) For each Java file, examine all code changes that occurred in the file through commits on the repository.
2) For each version of each Java file, make lists of methods[6] declared in the file by using our tool, JavaMethodExtractor.
3) For each method appearing at each version of each Java file, check if there was a change from the previous version: extract the code fragments corresponding to both the method and its previous version, and compare those fragments.
4) For each modified version of each method, extract all local variables by using our tool, JavaVariableScopeExtractor, and record their names, scopes and types.

Through the above four steps, we obtain the change history of each Java method and their local variable data. Then we analyze the data of each product with the following procedure:

1) Classify the set of local variables into four subsets according to the type categories (i)–(iv) defined in Sect. III-B1.

[4]https://github.com/{elastic/elasticsearch, alibaba/fastjson, google/guava, libgdx/libgdx, prestodb/presto, ReactiveX/RxJava}

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

[6]Constructors are included, but abstract methods are not included.

TABLE II
NUMBER OF METHODS APPEARING IN THE LATEST VERSION OF SURVEYED OSS PRODUCTS.

	#methods having		
product	no variable	one or more variables	total
Elasticsearch	20,203	4,347	24,550
Fastjson	1,163	1,010	2,173
Guava	16,064	5,439	21,503
libGDX	28,283	7,223	35,506
Presto	17,146	4,236	21,382
RxJava	10,983	2,761	13,744
Total	93,842	25,016	118,858

2) For each local variable in each subset, make the corresponding feature vector.
3) For each type category, compute the mean vector in the corresponding subset of feature vectors. If the distribution of metric values is skewed, perform a log transformation on the values before computing the mean vector.
4) For each local variable (each feature vector), compute the Mahalanobis distance from the mean vector of the corresponding type category: the Mahalanobis distance is the abnormality of the corresponding local variable.
5) Categorize methods into three groups: M_0, M_1 and M_2. M_0 is the set of methods having only low-level abnormal local variable(s); M_2 is the set of methods having a high-level abnormal local variable; M_1 is the set of other methods. Then, compare those three sets of methods in terms of the rates of change-prone methods; we consider a method whose change count is greater the third quantile (75% percentile) of all methods to be a change-prone method.

C. Results

As a result of our data collection, 118,858 methods were found in the surveyed products (see Table II). Since we are conducting an analysis focusing on local variables, our dataset consists of 25,016 methods which have a local variable.

1) Names Given to Local Variables:

For all products, the following trends were commonly observed: 1) the majority of local variables are of primitive or reference types; 2) arrays are the minority. Table III summarizes the frequently-appearing names of local variables ranked in the top 10 of declaration count[7] for each type category, where

[7]When two or more variables with the same name but different scopes appeared in a method, we counted them independently.

TABLE III
FREQUENTLY APPEARING NAMES.

type	variable names (#products)
(i) primitive	`i` (6), `index` (5), `size` (4), `result` (3), `value` (3), `count` (2), `offset` (2), `length` (2), `len` (2), `j` (2), `c` (2), `n` (2)
(ii) primitive array	`bytes` (5), `chars` (4), `array` (3), `values` (2), `buf` (2), `buffer` (2), `data` (2), `indexes` (2)
(iii) reference	`value` (6), `builder` (3), `entry` (3), `other` (2), `field` (2), `result` (2), `that` (2)
(iv) reference array	`array` (3), `files` (3), `values` (2), `items` (2), `result` (2), `fields` (2), `parameters` (2)

TABLE IV
DISTRIBUTIONS OF CHARACTER COUNTS, WORD COUNTS AND SCOPE OF LOCAL VARIABLE.

(a) character counts of local variable name

product	min	25%	50%	75%	max
(i) primitive					
Elasticsearch	1	3	**6**	10	35
Fastjson	1	2	**5**	7	30
Guava	1	1	**4**	8	30
libGDX	1	1	**4**	7	27
Presto	1	4	**7**	10	32
RxJava	1	1	**2**	5	23
(ii) primitive array					
Elasticsearch	1	4	**5**	10	19
Fastjson	1	4	**5**	7	26
Guava	1	5	**5**	7	22
libGDX	1	5	**6**	8	20
Presto	2	6	**9**	13	20
RxJava	—	—	—	—	—
(iii) reference					
Elasticsearch	1	5	**7**	11	33
Fastjson	1	5	**7**	10	30
Guava	1	5	**7**	9	30
libGDX	1	4	**6**	8	24
Presto	1	5	**8**	12	40
RxJava	1	1	**3**	5	31
(iv) reference array					
Elasticsearch	1	5	**7**	10	22
Fastjson	1	5	**7**	12	21
Guava	1	5	**6**	10	21
libGDX	1	5	**7**	9	21
Presto	4	6	**9**	12	24
RxJava	1	1	**1**	5	16

(b) word counts of local variable name

product	min	25%	50%	75%	max
(i) primitive					
Elasticsearch	1	1	**1**	2	8
Fastjson	1	1	**1**	2	5
Guava	1	1	**1**	2	7
libGDX	1	1	**1**	2	6
Presto	1	1	**1**	2	5
RxJava	1	1	**1**	1	4
(ii) primitive array					
Elasticsearch	1	1	**1**	1	3
Fastjson	1	1	**1**	1	4
Guava	1	1	**1**	1	5
libGDX	1	1	**1**	2	3
Presto	1	1	**2**	2	4
RxJava	—	—	—	—	—
(iii) reference					
Elasticsearch	1	1	**1**	2	7
Fastjson	1	1	**1**	2	5
Guava	1	1	**1**	2	5
libGDX	1	1	**1**	2	5
Presto	1	1	**1**	2	6
RxJava	1	1	**1**	1	5
(iv) reference array					
Elasticsearch	1	1	**1**	2	3
Fastjson	1	1	**1**	2	4
Guava	1	1	**1**	2	4
libGDX	1	1	**1**	2	5
Presto	1	1	**1**	2	4
RxJava	1	1	**1**	1	2

(c) scope of local variable

product	min	25%	50%	75%	max
(i) primitive					
Elasticsearch	2	5	**9**	22	354
Fastjson	2	9	**22**	47	422
Guava	1	4	**6**	11	80
libGDX	1	4	**10**	26	479
Presto	2	4	**8**	15	142
RxJava	2	9	**19**	39	181
(ii) primitive array					
Elasticsearch	2	4	**6**	10	59
Fastjson	2	3	**8**	31	167
Guava	2	5	**7**	15	56
libGDX	2	5	**10**	21	478
Presto	2	4	**8**	17.5	74
RxJava	—	—	—	—	—
(iii) reference					
Elasticsearch	2	4	**9**	23	356
Fastjson	1	5	**12**	36	424
Guava	1	3	**6**	11	135
libGDX	1	4	**9**	23	475
Presto	2	3	**7**	16	161
RxJava	2	5	**11**	26	236
(iv) reference array					
Elasticsearch	2	6	**14**	25	85
Fastjson	2	8	**20**	34	405
Guava	2	4.5	**8**	14	114
libGDX	2	5	**10**	18	212
Presto	2	6	**8**	17.5	76
RxJava	2	6	**12**	21	239

the number in parentheses signifies the number of products in which the name is ranked in the top 10. For example, name `i` of category (i) is a frequently-appearing name common to all six products; name `builder` of category (iii) is a frequently appearing name in three out of the six products.

From Table III, we can see that different type categories have different tendencies in their popular names. While names of (i) primitive type are the ones representing scalar quantities, names of (iii) reference type tend to be the ones having richer meanings. For both array types (ii) and (iv), the plural form of a word (e.g., `values`) or `array` is often used as the name of a local variable. While single-character names also appear in category (i), the majority of the names are single English words or their abbreviated forms. As these results show, popular names are short regardless of their type.

2) Distribution of Local Variables' Features:

Table IV shows the distributions of the above three properties of local variables. In the table, the medians are shown in boldface to ease capturing their trends. The results of category (ii) in RxJava are omitted due to the small number of samples.

At first, we see Table IV(a) showing the character count of variable's name. For category (i), while there are more single-character names than other type categories, the majority of character counts are around 2–7. For the remaining categories, the majority of lengths are around 5–9. That is to say, names in these categories tend to be longer than category (i).

Secondly, we focus on the word counts (Table IV(b)). Most results are common to all categories: the majority of names are single-word ones. However, there are compound names using two or more words in more than 25% of the variables. The difference of type category has less impact on the word count.

Next, we look at variables' scopes (Table IV(c)). Among all type categories, there are not so large differences in scope: most results are between a few lines and about 20 lines. The results may depend on the method's size.

Furthermore, we examined the Spearman rank-correlation coefficients among the above three properties as well. Table V shows the Spearman rank-correlation coefficients, where the strong correlations (> 0.6) are shown in boldface. In the table, strong positive correlations appear only in pairs of character count and word count. That is to say, there does not seem to be a tendency for programmers to give longer names to local variables having wider scopes in the real programs.

The above results showed that the majority of local variables have non-compound short names comprising of a few or less characters, with narrow scopes shorter than about 20 lines. Such features seem to match with the preferred features of local variables mentioned in many coding conventions.

3) Local Variables' Abnormalities vs. Change-Proneness of Methods:

We quantified local variables' abnormalities by the Mahalanobis distance. To examine the impact of the abnormality on quality of code, we classified the methods into the following three groups and compared their rates of change-prone methods: $M_0 = \{\ m \mid \mathrm{ma}(m) < 1\ \}$; $M_1 = \{\ m \mid 1 \leq \mathrm{ma}(m) < 2\ \}$; $M_2 = \{\ m \mid \mathrm{ma}(m) \geq 2\ \}$, where m is a method and $\mathrm{ma}(m)$ is the *maximum* abnormality of local variable(s) in m. Originally, we considered two types of metrics: the *maximum*

TABLE V
CORRELATIONS AMONG FEATURES.

product	char. vs. word	char. vs. scope	word vs. scope
(i) primitive			
Elasticsearch	**0.814**	0.276	0.205
Fastjson	**0.720**	0.268	0.166
Guava	**0.745**	0.392	0.294
libGDX	**0.667**	0.229	0.068
Presto	**0.833**	0.147	0.131
RxJava	0.464	0.257	0.063
(ii) primitive array			
Elasticsearch	**0.705**	0.171	0.158
Fastjson	0.483	0.134	0.474
Guava	**0.690**	0.199	0.234
libGDX	**0.680**	0.208	0.040
Presto	**0.813**	0.272	0.103
RxJava	—	—	—
(iii) reference			
Elasticsearch	**0.814**	0.098	0.094
Fastjson	**0.785**	0.140	0.155
Guava	**0.713**	0.128	0.124
libGDX	0.575	0.012	0.184
Presto	**0.836**	0.127	0.147
RxJava	0.427	0.077	−0.112
(iv) reference array			
Elasticsearch	**0.768**	0.164	0.039
Fastjson	**0.839**	0.095	−0.031
Guava	**0.778**	0.251	0.256
libGDX	**0.675**	0.082	0.047
Presto	**0.863**	−0.082	−0.229
RxJava	0.303	0.275	0.124

of abnormalities in a method and the *total* of abnormalities in a method. The maximum type focuses on the most abnormal variable. That is to say, our main focus is on the presence of abnormal variable. On the other hand, the total type takes into account abnormalities of all variables, so it seems to be an overall evaluation of abnormalities within a method. While both types of metrics have valid points, we also considered the influence by the method size, i.e., the question if LOC is a confounding factor. This is because it is generally true that a larger method is more change-prone. Thus, we examined the Spearman rank correlation coefficients between methods' LOC values and metric values of both two types. Table VI shows the results. In consequence, we adopted only the "maximum" type because the "total" type seems to be correlated with LOC.

Figure 1 shows the scatter diagram of the number of method changes and those methods' abnormalities (ma values). In the figure, the horizontal red line expresses the threshold which discriminates the change-prone methods corresponding to frequently-changed ones (whose change counts are greater than the third quantile); the vertical blue lines signify $ma(m) = 1$ and $ma(m) = 2$, respectively. That is to say, the scatter diagram is divided into six partitions H_i, L_i (for $i = 0, 1, 2$) as shown in Fig. 2. The methods plotted in either H_0, H_1 or H_2 are defined to be change-prone methods. From Fig. 1, we can see that more change-prone methods are in H_1 or H_2 than H_0. In other words, more change-prone methods are included in sets M_1 and M_2 than M_0, and those methods tend to have abnormal local variables.

TABLE VI
CORRELATION COEFFICIENTS IN PAIRS: LOC VS. THE MAXIMUM ABNORMALITY AND LOC VS. THE TOTAL ABNORMALITY.

	correlation coefficient	
product	max	total
Elasticsearch	0.206	0.555
Fastjson	0.331	0.721
Guava	0.224	0.499
libGDX	0.082	0.581
Presto	0.137	0.438
RxJava	0.038	0.471

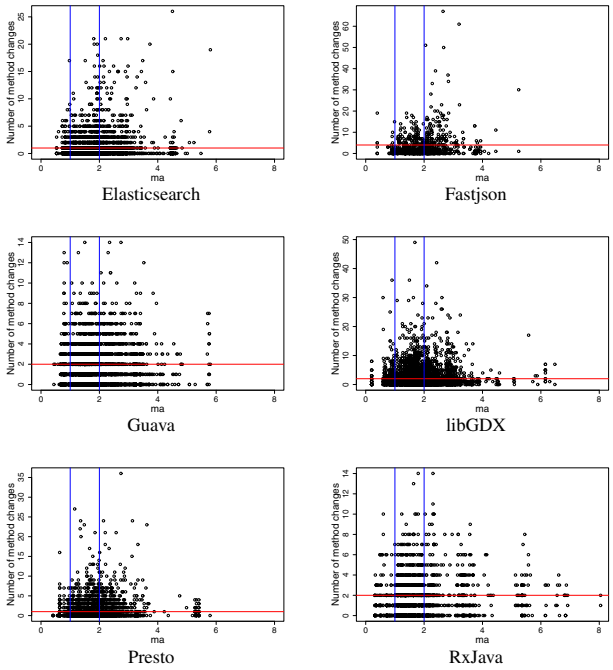

Fig. 1. Scatter diagram of the number of method changes vs. $ma(\cdot)$.

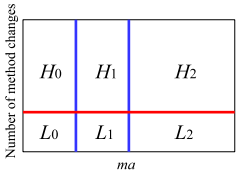

Fig. 2. Six partitions in Fig. 1.

Since the total numbers of methods in M_0, M_1 and M_2 differ each other, we examine the rates of change-prone methods in these three sets to compare their change-proneness. Table VII presents their rates which correspond to "(the number of methods plotted in H_i) / (the number of methods plotted in H_i or L_i)" (for $i = 0, 1, 2$). For all products, M_2 shows the highest rate of change-prone methods. In other words, the presence of an abnormal variable in a method is related to the method's change-proneness. The change-prone rate in M_2 is 1.2 – 2.5 times higher[8] than that in M_0 which is the set of

[8]Elasticsearch: $0.218/0.122 \simeq 1.8$, Fastjson: $0.262/0.104 \simeq 2.5$, Guava: $0.219/0.153 \simeq 1.4$, libGDX: $0.229/0.191 \simeq 1.2$, Presto: $0.252/0.175 \simeq 1.4$, RxJava: $0.240/0.123 \simeq 2.0$.

TABLE VII
RATES OF CHANGE-PRONE METHODS.

product	M_0	M_1	M_2
Elasticsearch	12.2% $(\frac{49}{401})$	16.7% $(\frac{436}{2606})$	21.8% $(\frac{297}{1360})$
Fastjson	10.4% $(\frac{7}{67})$	19.4% $(\frac{94}{485})$	26.2% $(\frac{124}{474})$
Guava	15.3% $(\frac{78}{509})$	13.6% $(\frac{471}{3473})$	21.9% $(\frac{351}{1601})$
libGDX	19.1% $(\frac{121}{635})$	14.6% $(\frac{661}{4519})$	22.9% $(\frac{453}{1975})$
Presto	17.5% $(\frac{63}{361})$	18.8% $(\frac{485}{2585})$	25.2% $(\frac{340}{1347})$
RxJava	12.3% $(\frac{29}{235})$	17.5% $(\frac{330}{1888})$	24.0% $(\frac{164}{682})$

TABLE VIII
RESULTS OF STATISTICAL TESTS EXAMINING DIFFERENCES IN THE RATES OF CHANGE-PRONE METHODS BETWEEN M_0 AND M_2.

product	χ^2	*df*	*p*-value	power of test
Elasticsearch	17.55	1	0.00003	0.995
Fastjson	7.06	1	0.0079	0.889
Guava	9.98	1	0.0015	0.917
libGDX	4.00	1	0.0456	0.552
Presto	9.16	1	0.0025	0.896
RxJava	13.72	1	0.0002	0.982

methods having only normal variable(s). For each product, we checked that the differences in rates between M_0 and M_2 using the χ^2 test, and all products showed statistically significant differences at $\alpha = 0.05$ (see Table VIII). While libGDX shows a smaller difference (19.1% vs. 22.9%) and a lower power of test (0.552), the remaining five products are tested with high powers of test (> 0.8), so the trend that "$M_0 < M_2$ in terms of change-proneness" seems to be supported.

However, the products do not always show monotonically increasing trends in the rates of change-prone methods. Thus, we cannot say that the evaluation of abnormality, $ma(m)$, has a strong correlation with the change-proneness of method m.

D. Discussions

1) RQ1 – What are the real trends of local variables in terms of their properties? (What are abnormal ones?):

We focused on local variables' properties: their names, scopes and types. From the results shown in Tables III and IV, we can say that variable types have impacts on their naming. Thus, our classification of variables according to their types is appropriate for analyzing the features of local variables.

Through the analysis of variables' names and scopes, we obtained the following tendencies:

1) For primitive type variables, programmers tend to give non-compound short names comprised of a few or less characters. It is not uncommon to see one or two-character names in primitive type variables. Major names are single English words, their abbreviated forms or their initial letters. It seems to dovetail with the fact that the normal length of English words is about 3 – 13 characters [10]. The normal length of scope is about 20 or less lines.
2) For reference type variables and arrays of primitive or reference type, programmers often give non-compound short names whose lengths are around a few characters but longer than the ones of primitive types. Their scopes are at the same level as the primitive type variables.
3) The length of a local variable's name has no correlation with its scope length.

Therefore, the majority of local variables in the real Java programming world seem to conform to the common coding conventions in terms of the length of names. However, scopes might be considered less important by many programmers.

From the above results, variables with longer or compound names or ones with wider scopes seem to be abnormal. Their abnormal levels are expressed by their Mahalanobis distance from the mean vector of features.

2) RQ2 – Is the presence of an abnormal local variable in a Java method related to the method's change-proneness?:

The abnormality analysis of local variables is worthwhile if we prove that an abnormal local variable is related to a poor quality of code. We analyzed one of measurable quality characteristics of Java methods—the change-proneness. As a result, a method having an abnormal local variable is 1.2 – 2.5 times more likely to be change-prone than the others, where the differences are statistically significant.

Although the maximum abnormality of local variables in a method seems to be related to change-proneness of the method, we have just reported the trend that change-prone methods are likely to have abnormal local variables. Toward a useful prediction of change-prone methods, we have to perform further analyses of local variables' abnormalities. Moreover, we need to clarify the reason why abnormal local variables are related to change-prone methods. Although there are concerns about abnormal local variables, e.g., a long name may cause a lack of readability [5], we have not yet proven whether an abnormal variable caused frequent code changes or not in the real. In order to see real effects of abnormal variables, we have to analyze code changes in more depth. Furthermore, authors of abnormal local variables—who introduced those variables—would also play an significant role in understanding the above tendency. These further analyses are our big challenges in the future.

E. Threats to Validity

We have analyzed large-scale Java OSS products. Since the notion of local variables is common to many modern programming languages, the difference in the language would not be a serious threat. However, in cases of small-scale OSS products or commercial ones, a few developers or organization-specific rules may have large impacts on properties of local variables. In order to enhance the generality of our results, we have to analyze the author and organization information as well.

Since we collected code changes in methods based on the identification of methods' signatures, we might miss some data when they were renamed or their arguments were changed. We must leverage more sophisticated technologies (e.g., Historage [11]) to reduce such a risk. Moreover, we did not see the details of code changes, so there might be changes which are independent of abnormal local variables, or local variables'

names and types might also be changed through the upgrades. We would tackle those problems as our important future work.

While the Mahalanobis distance is a useful measure to detect abnormal data, the decision of threshold value can be a threat to validity. We need to perform a further analysis to automatically decide an appropriate threshold in the future, for producing a better guideline of naming local variables.

V. Related Work

Binkley et al. [5] studied a relationship between the human short-term memory and the length of identifier through an empirical study with 158 programmers, and reported that a longer identifier tends to require a longer time to be understood. Thus, too long of a name seems to be related to a difficulty in code comprehension. Kawamoto et al. [12] analyzed identifiers used in Eclipse and NetBeans, and reported that a class having a longer identifier is more fault-prone. Their conclusions stating that a long name is related to poor quality are common to our results. However, identifiers in their studies include not only local variables but also classes and methods. Since names of classes or methods may be specified by the design documents, they would not be at the programmers' discretions.

Lawrie et al. [4] conducted an empirical study with 128 programmers to compare the understandability of a variable's name among three types: (a)fully-spelled word, (b) abbreviated one and (c)single character. They reported that the understandability showed (a)>(b)>(c), but the difference between (a) and (b) was not statistically significant. Liblit et al. [13] also conducted an empirical study with undergraduates, and reported similar trends with Lawrie et al. work [4]. That is to say, their studies proved that it is not necessary to give a long name to a variable. We examined such a trend while considering a variable's type and scope as well.

Aman et al. [14], [15] focused on local variables which have the maximum scopes in methods, and reported tendencies that longer names or compound names often appear in change-prone methods. While their work are our important previous work, they missed the consideration of variables' types. As we have seen, the type of variable often impacts on the naming. We introduced the notion of abnormality while considering the name, the scope and the type of variable in this paper.

VI. Conclusion and Future Work

We focused on local variables in Java methods. Since the naming of local variables is usually at programmer's discretion, we considered it may be a noteworthy point causing dispersion of code quality. We collected data on local variables from six popular large-scale OSS products, and evaluated their abnormalities in terms of their names, scopes and types by using the notion of the Mahalanobis distance. From the results, we obtained the following findings:

1) The trend of naming local variable differs according to the variable type: it is worthwhile to classify local variables in accordance with their types for accurate discussions on ways of naming local variables.
2) The majority of local variables have short names and narrow scopes; a name is often a word or its abbreviation.
3) Methods having abnormal local variables are about 1.2 – 2.5 times more likely to be change-prone than the others.

Java methods having deviant local variables do not tend to survive unscathed after their releases.

However, we have just reported the above trends in this paper, and we need further analyses of the impacts of abnormal local variables on the code quality in the future. We plan to perform various further analyses focusing on details of code changes, changes of abnormalities through upgrades, the meaning of words in variables' names, the author information and the conventions adopted by the development team. Moreover, we would like to produce an empirical-data-based guideline for the naming of local variables as well.

Acknowledgment

This work was supported by JSPS KAKENHI #16K00099. The authors would like to thank the anonymous reviewers for their helpful comments on an earlier version of this paper.

References

[1] B. W. Kernighan and R. Pike, *The practice of programming*. Boston, MA: Addison-Wesley Longman, 1999.
[2] Sun Microsystems, "Code conventions for the java programming language," http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html, 1999.
[3] Free Software Foundation, "Gnu coding standards," http://www.gnu.org/prep/standards/, 2016.
[4] 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.
[5] 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.
[6] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Boston, MA: Addison-Wesley, 2005.
[7] Google, "Google java style guide," https://google.github.io/styleguide/javaguide.html, 2017.
[8] D. Boswell and T. Foucher, *The Art of Readable Code*. Sebastopol, California: O'Reilly Media, 2011.
[9] G. Taguchi, S. Chowdhury, and Y. Wu, *The Mahalanobis-Taguchi System*. NY: McGraw-Hill, 2001.
[10] B. New, L. Ferrand, C. Pallier, and M. Brysbaert, "Reexaming the word length effect in visual word recognition: New evidence from the english lexicon project," *Psychonomic Bulletin & Review*, vol. 13, no. 1, pp. 45–52, Feb. 2006.
[11] H. Hata, O. Mizuno, and T. Kikuno, "Historage: fine-grained version control system for java," in *Proc. 12th Int'l Workshop on Principles of Softw. Evolution and 7th Annual ERCIM Workshop on Softw. Evolution*, Sept. 2011, pp. 96–100.
[12] 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.
[13] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proc. 18th Annual Psychology of Programming Workshop*, Sept. 2006.
[14] 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. 9th ACM/IEEE Int'l Symp. Empirical Softw. Eng. and Measurement*, Oct. 2015, pp. 50–53.
[15] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "Local variables with compound names and comments as signs of fault-prone java methods," in *Joint Proc. 4th Int'l Workshop on Quantitative Approaches to Softw. Quality and 1st Int'l Workshop on Technical Debt Analytics*, Dec. 2016, pp. 4–11.