Empirical Study of Fault-Prone Method's Name and Implementation: Analysis on Three Prefixes—Get, Set and Be

Sho Suzuki
Graduate School of Science and Engineering,
Ehime University
Matsuyama, Japan 790-8577
Email: s_suzuki@se.cite.ehime-u.ac.jp

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

Minoru Kawahara
Center for Information Technology
Ehime Unversity
Matsuyama, Japan 790-8577
Email: kawahara@ehime-u.ac.jp

***Abstract*—This paper focuses on the relationship between Java method's first word (prefix) and its implementation, for predicting fault-prone method. As a pilot study of the focused way of analysis, this paper analyzes three major prefixes of methods' names: "get", "set" and "be." The empirical study in this paper collects many data of methods' names and implementation features and analyzes abnormality of relationship between methods' names and their implementation features. Then, the study examines the relationship between a method's abnormality and its fault-proneness in eight open source software products. The empirical results prove the usefulness of focusing on the gap between methods' names and their implementation features in predicting fault-prone methods.**

I. Introduction

Effective and efficient quality management is essential to successful software development and maintenance. Many methods and tools for supporting quality managements have been studied and developed in the past. Especially, the fault-prone module prediction is one of the most useful topics. By evaluating fault-proneness of modules before reviewing and testing them, we can preferentially allocate our human resources (efforts) to suspicious modules which may have latent faults [1].

Most previous studies for predicting fault-prone modules are based on code metrics or process metrics (e.g., [2], [3], [4], [5], [6]). While those studies have been making remarkable contributions to the software engineering community, there have been a few studies focusing a semantic aspect of code—"name" of identifier: Lawrie et al. [7] presented an empirical report showing an impact of the way of naming identifiers on the source code understandability; Arnaoudova et al. [8] introduced linguistic anti-patterns of misleading pairs of method's (field's) name and implementation. However, there has not been a study on the relationship between such a semantic aspect and fault-proneness of module to the best of our knowledge. Thus, toward a more sophisticated fault-prone module prediction, we perform a fundamental investigation on relationships of Java methods' names to their implementations in this paper.

The remainder of this paper is organized as follows: Section II describes the way of naming methods in Java and the relationship between a method's name and its implementation, along with our related work. Section III reports our preliminary analysis. Section IV explains our empirical study, and presents the results and our discussions. Finally, Section V gives our conclusion and future work.

II. Method's Name and Implementation in Java

A. Method's name

When a method is developed, the developer has to give it a name which expresses its behavior. Under the limitation of language specification, developers can give any names to their methods. In the case of Java, the following conditions must be satisfied: a name is not a keyword, a boolean literal or an empty literal; not a symbol[1]; not started with a number.

Although any names satisfying the above three conditions are allowed to use, there are recommendations and conventions for naming methods (e.g., [9], [10]). For instance, the lower camel case is recommended for the name notation. In this notation, all words except for the first one have to start with a uppercase letter (e.g., `isJavaIdentifierPart`).

Moreover, there are also the following recommendations in regard to the words to be used: all words used in a method's name should be English words or their abbreviations; a method's name should be a verb or verb phrase; a method's name should be associated with the method's behavior; a pair of opposite words should be used for naming a pair of methods having opposite behaviors each other. For example, a name of a method which returns a value should start with "get" like `getBytes`, and a name of a method which sets a value to an object instance should start with "set" like `setSeed`. In order to give symmetric names to a pair of methods having opposite tasks, pairs (`get`, `set`), (`add`, `remove`) and (`open`, `close`) are often used in their naming.

[1]But an underscore "_" and dollar sign "$" are exceptionally available.

B. Relationship between Method's name and implementation: Related Work

There have been studies focusing on a relationship between a method's name and its implementation. Høst et al. [11] introduced the notion of regularity between a method's name and its implementation, and they summarized representative regularities as a phrase book [12]. Their phrase book presents the trends of methods in terms of their names and implementation features, which is a concrete collection of methods rather than lists of generalized naming recommendations given in coding conventions. Moreover, Høst et al. defined a distortion of method's name—a mismatch between the name and the implementation—as "naming bug," and recommended to rename the method or to change the implementation. Arnaoudova et al. [8] also concerned poor quality caused by an improper combination of names and implementations in methods. Kashiwabara et al. [13] created association rules in regard to pairs of methods' names and implementations, and proposed an automated system for suggesting a suitable first word of name for a method. While all of the above work are useful for evaluating or improving the quality of code, the impact of the gap between method's name and implementation on the fault-proneness has not been examined in the past—this is our research motivation.

III. Preliminary Analysis Focusing on First Word of Method's Name

In order to evaluate a distortion of method's name, we require baselines in regard to "normal" relationships between methods' names and implementation. Thus, we conducted a preliminary analysis of data to understand the real trends of methods in terms of their names and implementation features.

To ensure a generality of our empirical results, we used Qualitas Corpus created and published by Ewan et al. [14]. This corpus consists of 112 open source software data including their source files, binary files and so on. In this study, we developed a data collection tool based on JDT [15], and collected 18 boolean features shown in Table I from each method included in the latest version of source files in the corpus. Then, we categorized the results in accordance with the first words of their methods' names.

Figure 1 shows a part of the results corresponding the top 3 words of names—`get`, `set` and `be`[2]: for each word, each bar chart presents the rate of methods which are "true" at the corresponding feature.

From Fig. 1, we found the following features are common to three major named methods: they tend

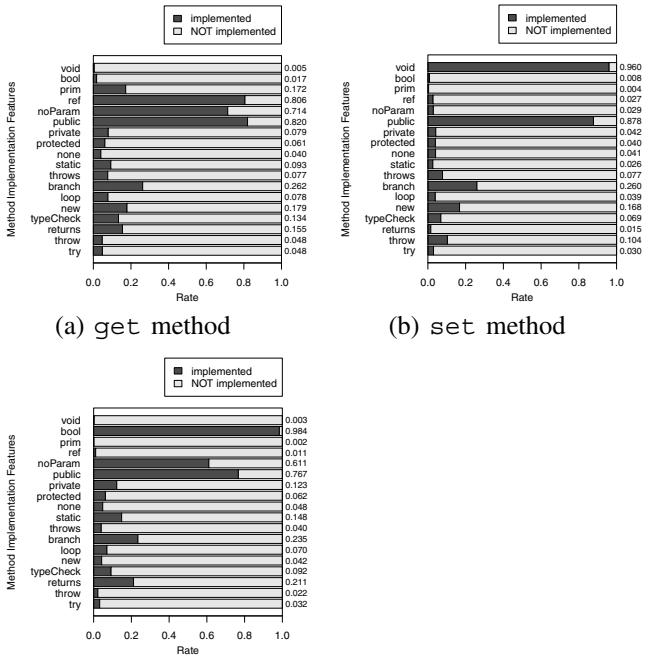
1) to be `public` method,
2) to have neither branch or loop structures,
3) not to create new object instance, and
4) not to throw an exception.

On the other hand, their features in regard to their return types and parameters showed difference according to the first words of their names:

[2]Word `be` corresponds to a be verb such as `is`, `are`, etc.

TABLE I
Collected features from each Java method.

feature	description (each value is either true or false)
void	is the return type `void`
bool	is the return type `boolean`
prim	is the return type a primitive
ref	is the return type a reference
noParam	has no parameters
public	has `public` access modifier
private	has `private` access modifier
protected	has `protected` access modifier
none	has no access modifier (i.e. package private)
static	has `static` modifier
throws	has thrown exceptions in `throws`
branch	has branch structure (e.g. `if` statement)
loop	has loop structure (e.g. `for` statement)
new	does create new object instance
typeCheck	has cast or `instanceOf` expression
returns	has some `return` statements
throw	dose throw an exception by itself
try	has `try` statement



(a) `get` method (b) `set` method

(c) `be` method

Fig. 1. Implementation features of methods whose names start with `get`, `set` or `be`.

1) `get` methods tend to have no parameter and return a value of reference type;
2) `set` methods tend to have one or more parameters and return nothing (`void`);
3) `be` methods tend to have not parameter and return a boolean value.

Based on the above results, we conduct our empirical study on the distortion of methods' names in the following section.

IV. Empirical Study

In this study, we evaluate a distortion of method's name (naming bug) and examine its impact on fault-proneness of

the method: we tackle the following research questions (RQs).

RQ1: Is the naming bug noteworthy in predicting fault-prone Java methods?

RQ2: Which implementation features are useful in discriminating faulty methods from non-faulty ones?

□

Notice that such a distortion does not always mean a wrong name because it might be caused by improper implementation features. In more precise words, a distortion signifies a gap between the name and the set of implementation features.

A. Data Collection

In this study, we collect data from eight open source software (OSS) products shown in Table II. We selected them since their source programs are written in Java and have been managed with Git. The former reason is from the limitation of the data collection tool we developed, and the latter reason is to use the Historage [16] for our fine-grained analysis of code changes.

To discuss source code quality at the method level, we have to track method's changes in their version control repository. For this reason, we decided to use Historage which is a fine-grained version control system developed by Hata et al. A Historage repository allows us to track changes with respect to class, method or field. A Git repository can be converted to Historage one through the Historage-related tool, Kenja[3].

Our data collection is performed in the following procedure.

1) Get clones of repositories:
   Make copies of Git repositories on our local disk.
2) Convert Git repositories to Historage ones:
   Convert Git repositories gotten in step 1 to Historage repositories by Kenja
3) Extract histories of each method's changes from Historage repository:
   Extract method-change histories from corresponding Historage repositories
4) Extract the first word of each method's name and get its base form:
   Extract the first word of each method's name by splitting the name into words with the following regular expressions:
   - _(underscore)
   - `(?<=[a-z])(?=[A-Z])`
   - `(?<=[A-Z])(?=[A-Z][a-z])`
   - `(?<=\D)(?=\d)|(?<=\d)(?=\D)`

   Then, convert the word to its base form by using the GENIA tagger[4]
5) Extract each method's implementation features:
   Extract each method's implementation features as mentioned in Sect. III.

□

In our data collection, we excluded (1) constructors, (2) methods in an anonymous class, and (3) methods deployed in a test case directory. The reasons are: (1) names of constructors totally depend on their class names; (2) methods in an anonymous class often have no names; (3) methods deployed in a test case directory are developed for testing, so an analysis of their fault-proneness seems to be out of our scope in this paper.

B. Data Analysis

1) Method Classification and Comparison of Fault Rates:

For RQ1, we perform our analysis in the following steps.

1) Extract methods whose names start with `get`, `set` or `be`; create three sets of methods corresponding to their first words.
2) Divide each set of methods into two subsets: the set of normal methods vs. the set of abnormal methods.
   Take an example of the feature "void" of `get` methods. From the results of preliminary analysis shown in Fig. 1(a), we can see that the rate of `get` methods having no return value (`void`) is 0.005. Therefore, if a `get` method has a return value, it is normal; otherwise it is abnormal. We will judge their abnormality with threshold 0.05: if a feature's true rate is less than 0.05, we consider the feature of a normal method should be false; if a feature's true rate is greater than 0.95, we regard the feature of a normal method should be true. In this paper, we call such an abnormal case "fatal naming bug," and consider a method to be abnormal if it has a fatal naming bug.
3) Compare the rates of faulty methods (fault rates).
   Compare the fault rates between the set of normal methods and that of abnormal methods; we decide whether a method is faulty or not by checking their commit logs.

2) Which implementation features are useful in discriminating faulty methods from non-faulty ones?:

For RQ 2, we examine the implementation features in the following procedure for each set of `get` methods, `set` ones and `be` ones:

1) Divide the set of methods into two subsets: the subset of non-faulty methods and the subset of faulty ones.
2) For each subset, compute the true rates of implementation features.

[3]https://github.com/niyaton/kenja

[4]http://www.nactem.ac.uk/GENIA/tagger/

TABLE II
SURVEYED OSS PRODUCTS.

product	# of commits	investigation period
Eclipse Checkstyle Plug-in	993	2003-05 ∼ 2016-11
FreeMind	1,062	2011-02 ∼ 2016-08
Dr Java	3,466	2001-06 ∼ 2014-10
PMD	9,079	2002-06 ∼ 2016-12
react-native	6,077	2015-01 ∼ 2016-04
retrofit	1,313	2010-09 ∼ 2016-04
Universal Image Loader	1,025	2011-11 ∼ 2016-01
RxJava	4,885	2012-03 ∼ 2016-12

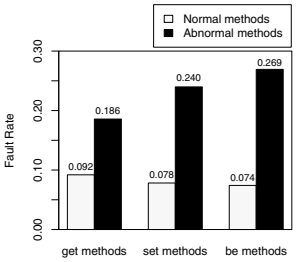
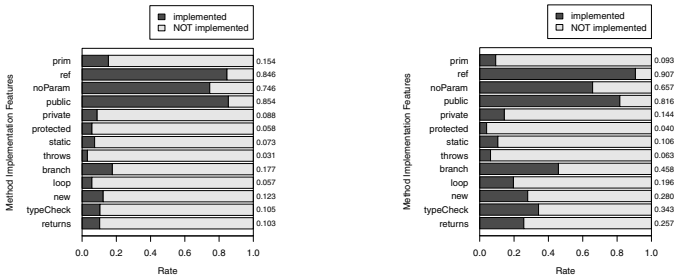


Fig. 2. Comparison of fault rates.



(a) non-faulty methods (b) faulty methods

Fig. 3. True rates of features in `get` methods.

3) Compare the true rates of implementation features between the set of non-faulty methods and that of faulty ones, in a statistical way.

C. Results

1) Is the naming bug noteworthy in predicting fault-prone Java methods?:

Figure 2 shows the fault rates in the set of normal methods and that of abnormal methods having a fatal naming bug, for each of `get` methods, `set` ones and `be` ones. From this figure, we can say that an abnormal method having a fatal naming bug is more likely to be faulty than a normal method: the risks of being faulty are about 2 times, 3 times and 3.6 times higher in `get` methods, `set` methods and `be` methods, respectively. Therefore, fatal naming bugs would be noteworthy in predicting fault-prone Java methods.

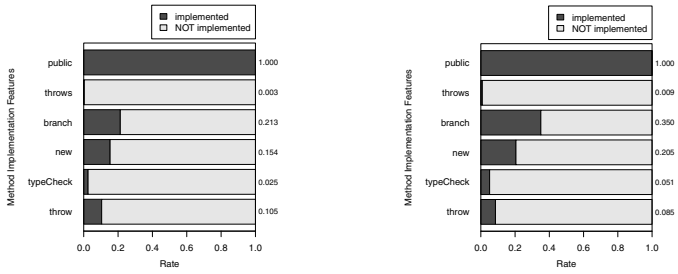
2) Which implementation features are useful in discriminating faulty methods from non-faulty ones?:

Figure 3 presents the true rates of implementation features in (a) non-faulty `get` methods and (b) faulty ones, respectively. Similarly, Figs. 4 and 5 presents those bar charts for `set` methods and `be` methods, respectively. Tables III, IV and V compare the true rates of each feature in non-faulty methods and in faulty ones, and the results of statistical tests (p values) examining the differences between their true rates, for `get` methods, `set` ones and `be` ones, respectively.

For `get` methods, faulty methods are more likely to have the following eight features: 1) having a branch structure (e.g., `if` statement), 2) having a cast expression or `instanceOf` operator, 3) having multiple `return` statements, 4) having a loop structure (e.g., `for` statement), 5) having no parameter, 6) being private method, 7) being static method, and 8) having `throws` statement.

TABLE III
TRUE RATES OF FEATURES AND RESULTS OF STATISTICAL TESTS (GET METHOD).

feature	(a)non-faulty	(b)faulty	(b) − (a)	p value
prim	0.154	0.093	−0.061	0.002
ref	0.846	0.907	0.061	0.002
noParam	0.746	0.657	−0.089	< 0.001
public	0.854	0.816	−0.038	0.050
private	0.088	0.144	0.056	< 0.001
protected	0.058	0.040	−0.018	0.187
static	0.073	0.106	0.033	0.026
throws	0.031	0.063	0.032	0.001
branch	0.177	0.458	0.281	< 0.001
loop	0.057	0.196	0.139	< 0.001
new	0.123	0.280	0.157	< 0.001
typeCheck	0.105	0.343	0.238	< 0.001
returns	0.103	0.257	0.154	< 0.001



(a) non-faulty methods (b) faulty methods

Fig. 4. True rates of features in `set` methods.

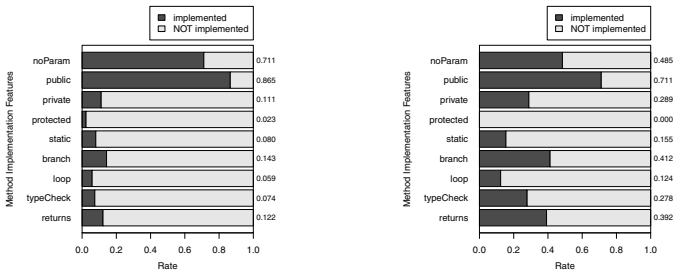
For `set` methods, the presence of a branch structure is only the feature which is more frequently observed in faulty methods than non-faulty ones.

For `be` methods, faulty methods tend to have the following five features: 1) having a branch structure, 2) having a cast expression or `instanceOf` operator, 3) having multiple `return` statements, 4) being a private method, and 5) having a parameter. From the above trends, we can say that normal `be` methods are usually public, instance methods which have no parameter and no branch structure, do not special data processing, and do not have multiple `return` statements.

Moreover, as shown in Fig. 2, the difference of fault rates between the set of non-faulty `be` methods and that of faulty

TABLE IV
TRUE RATES OF FEATURES AND RESULTS OF STATISTICAL TESTS (SET METHOD).

feature	(a)non-faulty	(b)faulty	(b) − (a)	p value
public	1.000	1.000	0.000	NA
throws	0.003	0.009	0.006	0.903
branch	0.213	0.350	0.137	< 0.001
new	0.154	0.205	0.051	0.184
typeCheck	0.025	0.051	0.026	0.179
throw	0.105	0.085	−0.020	0.625



(a) non-faulty methods (b) faulty methods

Fig. 5. True rates of features in be methods.

TABLE V
TRUE RATES OF FEATURES AND RESULTS OF STATISTICAL TESTS (BE METHOD).

feature	(a)non-faulty	(b)faulty	(b) − (a)	p value
noParam	0.711	0.485	−0.226	< 0.001
public	0.865	0.711	−0.154	< 0.001
private	0.111	0.289	0.178	< 0.001
protected	0.023	0.000	−0.023	0.250
static	0.080	0.155	0.075	0.019
branch	0.143	0.412	0.269	< 0.001
loop	0.059	0.124	0.065	0.021
typeCheck	0.074	0.278	0.204	< 0.001
returns	0.122	0.392	0.270	< 0.001

be methods is larger than the cases of get methods and set ones. Thus, be methods are possibly easier to be more complicated. While get methods and set ones usually play the roles of accessors to values, be methods may be required more complex processes in regard to judgments involved in many other data or status of objects.

D. Discussions

1) get method:

Figure 6(a) shows an example of method which has no fatal naming bug. This method is included in ImageSize.java of Universal Image Lorder; the class of this method has two fields height and width. This method returns the value of width, which looks a normal get method.

On the other hand, Fig. 6(b) shows an method having a fatal naming bug on the return type, which is included in MindMapMapModel.java of FreeMind. This method does not return anything, but writes strings via Writer object. From name getXml, many programmers may assume that it returns a data of XML format. That is to say, this implementation is confusable and may cause unexpected faults.

2) set method:

Figure 7(a) shows an example of a normal method which has no fatal naming bug. This method appears in ImportDeclaration.java of DrJava; the class has a string field name. Then, this method sets a value to the field unless a parameter value is null, which seems to be one of general set methods.

On the other hand, the method shown in Fig. 7(b) is abnormal, which is included in LanguageVersionDiscoverer.java of PMD. The strangest point is at its return type (reference type).

```
public int getWidth() {
  return width;
}
```

(a) A normal get method (ImageSize.java of Universal Image Lorder).

```
/**
* writes the content of the map to a writer.
*
* @throws IOException
*/
public void getXml(Writer fileout, boolean saveInvisible,
  MindMapNode pRootNode) throws IOException {
  fileout.write("<map ");
  fileout.write("version=\"" + FreeMind.XML_VERSION + "\"");
  fileout.write(">\n");
  fileout.write("<!-- To view this file, download free mind mapping software
        FreeMind from http://freemind.sourceforge.net -->\n");
  pRootNode.save(fileout, this.getLinkRegistry(), saveInvisible, true);
  fileout.write("</map>\n");
  fileout.close();
}
```

(b) An abnormal get method (MindMapMapModel.java of FreeMind).

Fig. 6. Examples of get methods.

As shown in Fig. 1(b), it is really rare that a set method has such a return value. As described in its Javadoc (see Fig. 7(b)), this method sets a given value as the current language information and returns previous language information. However, in this case, these behaviors can be divided into two sub methods, i.e., this method should be refactored. In the future, we would like to automatically recommend a code refactoring to such a method, based on our research results.

3) be method:

Figure 8(a) shows an example of normal method included in NodeAdapter.java of FreeMind. This method judges whether the state of Filter instance is visible or not. In the concrete, this method first gets Filter instance by method invocations, then performs the logical operation using the result. As shown in Fig. 8(a), it returns a boolean value, does not have any

```
/**
* Sets the package name
* @exception IllegalArgumentException if s is null
*/
public void setName(String s) {
  if (s == null) throw new IllegalArgumentException("s == null");
  name = s;
}
```

(a) A normal set method (ImportDeclaration.java of DrJava).

```
/**
* Set the given LanguageVersion as the current default for it's Language.
*
* @param languageVersion
*            The new default for the Language.
* @return The previous default version for the language.
*/
public LanguageVersion setDefaultLanguageVersion(LanguageVersion
      languageVersion) {
  LanguageVersion currentLanguageVersion = languageToLanguageVersion.put(
        languageVersion.getLanguage(),
         languageVersion);
  if (currentLanguageVersion == null) {
    currentLanguageVersion = languageVersion.getLanguage().getDefaultVersion
          ();
  }
  return currentLanguageVersion;
}
```

(b) An abnormal set method (LanguageVersionDiscoverer.java of PMD).

Fig. 7. Examples of set methods.

```
public boolean isVisible() {
  Filter filter = getMap().getFilter();
  return filter == null || filter.isVisible(this);
}
```

(a) A normal be method (NodeAdapter.java of FreeMind).

```
/**
* True if there is a shared map at present, false otherwise. If this value
* is false, the sender will ignore requests to send Freemind commands.
*
* @param shared
*/
public void isMapShared(boolean shared) {
  mapShared = shared;
}
```

(b) An abnormal be method (JabberSender.java of FreeMind).

Fig. 8. Examples of be methods.

parameter and complex implementations. This method would be one of representative be methods.

On the other hand, the method shown in Fig. 8(b) has a fatal naming bug on its return type. This method appears in JabberSender.java of FreeMind. According to its return type and Javadoc description, this method does not judge anything. Moreover, it sets a parameter value to field mapShared. In this case, we consider that the method should be set one; this method's name is confusable and may be troublesome.

E. Threat to Validity

There are concerns in regard to our data collection.

Since we used the Qualitas Corpus in our preliminary analysis to understand the real trends of methods in terms of their names and implementation features in Sect. III, there may be a kind of data bias. Moreover, we did not considered the size of data—the number of methods in each product. Thus, the preliminary results are possibly affected by a few large-scale software products. There is a similar threat to validity for the empirical results in Sect. IV. While we mitigated such threats by using a lot of methods from many products of various domains, replication studies would be required to prove the generality of our results.

While we excluded test programs, there might be miss filtered cases because we carried out the exclusion by using a simple pattern matching to their source path name. However, test programs are not usually mixed into the same directories as application programs, so the way of excluding testing code would not be serious threat to validity in our study.

We used Historage repositories to track method change histories, thus the reliability of code change data depends on the Historage technology. Since the Historage technology has been actively leveraged in the mining software repository community, we believe that it is not our threat to validity.

V. Conclusion

Toward yet another way of predicting faulty methods, we have focused on the gap between method's name and its implementation—a naming bug. As a pilot study, we analyzed major methods whose prefixes of names are "get," "set" and "be" in this paper.

In our empirical study with open source products, we classified the set of methods into the set of normal methods and that of abnormal methods in terms of fatal naming bug, and compared their fault rates. The results showed that the fault rates of abnormal methods are $2 - 3.6$ times larger than normal ones. Therefore, we can say that fatal naming bugs are noteworthy for fault-prone method prediction.

Moreover, we conducted a further analysis on which implementation features are useful in predicting faulty methods, and found remarkable points to be checked for each of get method, set method and be method, respectively. Those check points would be useful clues during their code reviews.

In the future, we plan to perform a further analysis on methods whose prefixes of names are not only "get," "set" and "be' but also other verbs in order to enhance the generality of our results and the accuracy of prediction.

References

[1] 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.

[2] T. M. Khoshgoftaar and N. Seliya, "Comparative assessment of software quality classification techniques: An empirical case study," *Empirical Softw. Eng.*, vol. 9, no. 3, pp. 229–257, Sept. 2004.

[3] 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.

[4] 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.

[5] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. ACM/IEEE 30th Int'l Conf. Softw. Eng.*, May 2008, pp. 181–190.

[6] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. 2013 Int'l Conf. on 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] V. Arnaoudova, M. D. Penta, G. Antoniol, and Y. G. Guhneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *Proc. 17th European Conf. Softw. Maintenance and Reengineering*, March 2013, pp. 187–196.

[9] Oracle, "The java language specification, java se 8 edition," http://docs.oracle.com/javase/specs/jls/se8/html/index.html, Feb. 2015.

[10] Google, "Google java style," https://google.github.io/styleguide/javaguide.html, Mar. 2014.

[11] E. W. Høst and B. M. Østvold, "The programmer's lexicon, volume i: The verbs," in *Proc. 7th IEEE Int'l Working Conf. Source Code Analysis and Manipulation*, Sept. 2007, pp. 193–202.

[12] ——, "The java programmers phrase book," in *Softw. Language Eng.*, ser. Lecture Notes in Computer Science, D. Gašević, R. Lämmel, and E. V. Wyk, Eds., vol. 5452. Springer Berlin Heidelberg, 2009, pp. 322–341.

[13] Y. Kashiwabara, T. Ishio, H. Hata, and K. Inoue, "Method verb recommendation using association rule mining in a set of existing projects," *IEICE Trans. Inf. & Syst.*, vol. E98-D, no. 3, pp. 627–636, Mar. 2015.

[14] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Softw. Eng. Conf.*, Dec. 2010, pp. 336–345.

[15] The Eclipse Foundation, "Eclipse java development tools (jdt)," http://www.eclipse.org/jdt/, Jan. 2017.

[16] 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*, 2011, pp. 96–100.