Recommending Verbs for Rename Method using Association Rule Mining

Yuki Kashiwabara*, Yuya Onizuka*, Takashi Ishio*, Yasuhiro Hayase†, Tetsuo Yamamoto‡, and Katsuro Inoue*
*Graduate School of Information Science and Technology, Osaka University, Japan
E-Mail: {k-yuki, y-onizuk, ishio, inoue}@ist.osaka-u.ac.jp
†Graduate School of Systems and Information Engineering, University of Tsukuba, Japan
E-Mail: hayase@cs.tsukuba.ac.jp
‡College of Engineering, Nihon University, Japan
E-Mail:tetsuo@cs.ce.nihon-u.ac.jp

***Abstract*—An identifier is one of the crucial elements for program readability. Method names in an object-oriented program are important identifiers because method names are used for understanding the behavior of the methods without reading a part of the program. It is well-known that each method name should consist of a verb and objects according to general guidelines. However, it is not easy to name methods consistently since each of the developers may have a different understanding of the verbs and objects used in the method names. As a first step to enable developers to name methods consistently and easily, we focus on the verbs used in the method names.**

In this paper, we present a technique to recommend candidate verbs for a method name so that developers can use consistent verbs for method names. Given a method, we recommend a list of verbs used in many other methods similar to the given method, by using association rules. We have extracted association rules from 445 OSS projects and applied these rules to two projects. As a result, the extracted rules could recommend the current verbs in the top 10 candidates for 60.6% of the methods covered by our approach. Furthermore, we have identified four meaningful groups of rules for verb recommendation.

I. Introduction

Identifier is one of the crucial elements for program readability [1]. Developers take a considerably long time to understand a program, if the identifiers poorly represent their roles in the program [2]. Since developers spend a lot of time on program reading [3], good identifiers are important to reduce the cost of software maintenance.

Method names are important identifiers because method names are used for understanding the behavior of the methods without reading the program. According to the general guidelines for an object-oriented program [4], a method should have a name representing its behavior. A method name generally consists of a verb and objects; this name is expected to represent the behavior of the method consistently. However, it is not easy for developers to choose a verb and objects consistently since each developer may have a different understanding of the verbs and objects used for method names. Only a few verbs such as `get`, `set`, and `test` are consistently used for representing the behavior of a method among developers.

A tool recommending good verbs for a method name is a first step to creating a tool recommending good method names to developers for rename method refactoring. As a previous work, Høst *et al.* investigated relationships between the behavior of methods and the verbs used in the method names [5]. They reported that there exist naming rules for 40 verbs. For example, methods named `find` often contain loops and use local variables. On the basis of the naming rules, they implemented a rule-based naming bug detection tool [6], [7]. Their tool acccurately points out inappropriate verbs used in method names. However, their tool is appropriate for naming bug detection against limited methods. There is no tool to recommend candidates of consistent names to many methods.

In this paper, we propose a technique to recommend candidates of verbs for a method name so that developers can consistently use various verbs. We assume that the behavior of a method is often characterized by identifiers such as method calls and field access in the method definitions. We extract the relationship between verbs used in method names and identifiers in method definitions from existing source files, by using association rule mining [8]. Using the extracted rules, we recommend candidates of verbs likely to be used as a part of a method name, *e.g.*, *if a method calls* `next`, `hasNext`, `iterator`, *and* `equals`, *then* `find` *is likely to be a verb representing the behavior*.

We have extracted association rules from 445 OSS projects written in Java and applied the rules to two OSS projects to evaluate whether the current verbs used in method names could be recommended or not. We regarded a verb already used in a target method name as the correct verb of the target method. As a result, we found that 92.1% of the considered methods have at least one rule recommending the correct verb and 60.6% of the methods have been found in the top 10 candidates for each method. In addition to the quantitative analysis, we analyzed what kind of association rules are used for recommending the verbs. We have identified four meaningful groups of rules for verb recommendation as follows: The first group of rules recommends the same verb as methods called in the method, *e.g.*, `add` for a method using another `add` method. The second group recommends verbs that are conceptually related to a certain word in the method, *e.g.*, `execute` for a method using an argument `command`. The third group recommends verbs related to a class definition, *e.g.*, `compare` for a method defined in a

`Comparable` class. The fourth group recommends verbs on the basis of the Java programming idioms, *e.g.*, `find` for a method using `iterator`. We expect these extracted rules to provide understandable verbs to developers.

The main contributions of this paper are:

- We have defined an application of association rule mining to extract relationships between verbs used in method names and identifiers in method definitions.
- We have shown that association rules extracted from OSS projects are applicable to the recommendation of candidate verbs for methods in different applications.
- We have shown that the technique can recommend correct verbs for 60.6% of the methods coverd by our approach.

The rest of this paper is organized as follows: Section II explains the background of our research. Section III describes our approach to recommending candidates verbs. Section IV shows the result of our experiment. Section V discusses threats to the validity of the proposed approach and the experiment. Section VI presents the conclusions and future work.

II. Related Work

A. Studies on Support of Naming Method

Høst *et al.* analyzed the relationship between the behavior of methods and the verbs used in the method names [5]. First, they split method names into verbs and objects. Secondly, for each verb, they analyzed the typical behavior of methods including the verb in their names. Finally, they identified the typical behavior for 40 concrete verbs. The following is a quote from their rules:

> find: Methods named `find` very often use local variables and contain loops. Furthermore, they often perform type-checking, and rarely return void.

On the basis of the above study, Høst *et al.* proposed a technique and a tool that alerts the naming bugs of methods to developers and that provides how to fix the naming bugs [6], [7]. They have defined negative rules between the behavior of methods and method names. A negative rule specifies that a target method has a naming bug, if the method follows the rule. Although the technique is highly accurate for the naming bug detection, developers can receive the tool support for a limited number of methods whose names are covered by 76 patterns using 64 verbs. In our work, we use association rule mining to cover a large number of verbs in source files for the renaming method, instead of the naming bug detection.

Hayase *et al.* created domain-specific dictionaries by collecting verb-oriented relations from identifiers appearing in source files [9]. We get the idea that we use identifiers to extract rules.

B. Association Rule Mining

Association rule mining [8] is a technique used for extracting association rules from a large number of *transactions*. Each transaction is a set of items. One association rule represents a fact that an item set frequently appears with another item set at the same time. An association rule is described by the following expression: $X \rightarrow Y$. X and Y are called the antecedent and the consequent, respectively. Both are subsets of a transaction.

There are two well-known definitions to measure the significance and the interest of the extracted rules: confidence and support. The confidence of $X \rightarrow Y$ indicates the ratio of the number of transactions involving $X \cup Y$ against the number of transactions involving X. The support of $X \rightarrow Y$ indicates the number of transactions in which all items in $X \cup Y$ appear at the same time. We consider that there is a correlation between X and Y, if the confidence and the support are higher than the certain thresholds.

Singer *et al.* applied association rule mining to the structural and behavioral attributes of methods called nano-patterns [10]. They reported several relationships among attributes, *e.g.*, most methods reading an array contain a loop in their definitions. While their approach is similar to our approach, we focus on the relationship between method names and the identifiers in the methods.

III. Proposed Approach

We recommend candidate verbs for a method name using association rule mining. The proposed approach consists of two steps: extracting rules between verbs used in method names and the identifiers in method definitions, and applying the rules to recommend verbs for a method name. We call the extracted rules **naming association rules**.

A. Extraction of Naming Association Rule

This step takes as input a set of Java source files as a training dataset. We create an AST-tree for each source file and extract a set of methods M from the source files. We exclude the following methods from M, because their verbs are well-known or determined by using the Java language.

`main` methods and constructors: These names are defined by a Java language specification.

methods defined in anonymous inner class: Most of the methods are inherited from parent classes.

`get` and `set` methods: Both "get" and "set" are well-known verbs for field access methods.

`test` methods: The verb "test" is also well-known for the JUnit testing framework.

`toString`, `hashCode`, and `equals` methods: These names are inherited from `java.lang.Object`.

We generate a set of transactions T from M, by translating each method $m \in M$ into a transaction $t(m)$ that is a set of elements, which are pairs of an identifier and its category. Each element is represented as "$category$:$name$," where $category$ denotes the category of the identifier and $name$ represents the text of the identifier. For example, if m calls a method `add` in the definition, $t(m)$ contains "call:add" as an element. We extract the following nine types of elements in a method m defined in class C as $t(m)$.

method-verb: A verb used in the name of m. To extract a verb from the method name, we have used OpenNLP[11], which is a natural language processing tool. We have considered six words `to, new, init, calc, cleanup`, and `setup` as verbs, because these words are often used as words similar to verbs in Java programs. We do not use stemming. We analyze similar verbs including synonyms individually.

class-name: Name of class C.

parent-class-name: Name of the parent class of C. We ignore `java.lang.Object` if it is not explicitly declared in the class.

interface-name: Name of an interface implemented by C. We extract names only if they are explicitly declared in the class. In other words, we ignore interfaces inherited from the parent class of C but not declared in C.

return-type: Return type of m.

argument-type: Type of an argument of m.

argument-name: Name of an argument of m.

field-name: Name of a field that is defined in class C and accessed in method m. We ignore fields defined in other classes including the parent class.

call: Name of a method directly called by m.

We ignore the names and the types of local variables because they tend to represent data manipulated in a method rather than actions in the method. Further, we ignore method signatures for a method called by m. For example, both `ArrayList.add` and `LinkedList.add` are regarded as the same element "call:add."

Fig. 1 shows how a source file is translated into transactions. The source file in Fig. 1 (a) includes two methods: `findName` and `addName`. Figs. 1 (b) and (c) represent $t($`findName`$)$ and $t($`addName`$)$, respectively.

We apply association rule mining to the transaction set T with four conditions for filtering rules. The first and the second conditions ensure that a rule recommends a verb.

1) The antecedent of a rule contains no method-verbs.

2) The consequent of a rule contains only one method-verb.

Hence, a naming association rule can be denoted as (X, v, c, s), where X denotes the antecedent; v, the consequent {method-verb:v}; c, the confidence; and s, the support. For example, if 100 methods whose verb is `add` and 80 of them call an `addAll` method in their method definitions, a naming association rule $(\{$call:addAll$\}$,add$, 0.8, 80)$ is extracted.

We use the following conditions for further filtering.

3) Antecedent $|X| \leq 4$.

4) Support $s \geq 20$.

The third condition extracts simpler rules to reduce the effort of the manual analysis of extracted rules. The fourth condition prevents naming association rules from overfitting.

B. Applying Rules to Recommend Verbs

In this step, we use a set of naming association rules R to recommend verbs for a given method m. We extract a transaction $t(m)$ from the method and select the applicable rules $Applicable(m)$ as follows.

$$Applicable(m) = \{(X, v, c, s) | X \subseteq t(m) \land (X, v, c, s) \in R\}$$

```
public class NameList implements Serializable{
 List<String> nameList;

 public String findName(String name){
   if (nameList.contains(name)) {
     return name;
   }
   return null;
 }

 public void addName(String name, int index){
   if(!nameList.contains(name)){
     nameList.add(index, name);
   }
 }
}
```

(a) source file

method-verb:	find
class-name:	NameList
interface-name:	Serializable
return-type:	String
argument-type:	String
argument-name:	name
field-name:	nameList
call:	contains

(b) the transaction extracted from `findName` method

method-verb:	add
class-name:	NameList
interface-name:	Serializable
return-type:	void
argument-type:	int
argument-name:	index
argument-type:	String
argument-name:	name
field-name:	nameList
call:	contains
call:	add

(c) the transaction extracted from `addName` method

Fig. 1. An example of translation from Java source file

We regard the consequent v of a rule in $Applicable(m)$ as a recommendation from the rule. If more than one rule recommends the same verb, we use the rule with the highest confidence c. We sort the recommended verbs in the descending order of their confidence values and provide the resultant list to developers.

IV. Evaluation

We have investigated the following two research questions to evaluate the proposed approach:

1) How many verbs can be recommended correctly?
2) Are naming association rules meaningful?

We extracted naming association rules from 445 OSS projects obtained from sourceforge.net [12], apache.org [13], and eclipse.org [14]. We applied the extracted rules to two OSS projects: ArgoUML and jEdit. Both are not included in the training dataset.

Table I presents an overview of the dataset and the target projects. #LOC denotes the number of lines in the source files. #Method indicates the number of methods in the source files, and #Analyzed represents the number of analyzed methods as described in Section III-A. The number of the extracted naming association rules is 1,475,419. The rules are extracted from 594,439 (77.8%) methods.

A. RQ1: How Many Verbs can Be Recommended Correctly?

To answer this research question, we have evaluated whether verbs currently used in a program can be recommended by the

TABLE I
OVERVIEW OF DATASET AND TARGET PROJECTS

Source Files	#LOC	#Method	#Analyzed
Training dataset	34,326,308	1,399,744	764,303
ArgoUML 0.28.1	367,052	15,008	6,651
jEdit 4.3.1	176,556	6,299	2,676

rules extracted from the training dataset or not. While there may be naming bugs in a program, we have regarded the verb of a method m as the correct result for the method m, because a few naming bugs were detected by the existing tool [6], [7].

We have computed the rank of the correct verb in the recommendation list for each method. Fig. 2 plotted the result. The horizontal axis represents the number of methods. The vertical logarithmic axis represents the rank of the concrete verb for a method. Since we computed the rank for each method, we sorted the methods in the ascending order of their rank values. The two vertical solid lines indicate the number of analyzed methods in the applications: 6651 in ArgoUML and 2676 in jEdit.

6,093 (91.6% of 6651) methods in ArgoUML and 2,500 (93.5% of 2676) methods in jEdit have at least one rule recommending the correct verb. The proposed approach recommended the correct verb in the top of a ranking for 1,841 (30.2% of 6,093) methods in ArgoUML and for 738 (29.5% of 2,500) methods in jEdit. If we recommend the top 10 candidates to developers, correct verbs are recommended for 3,781 (62.0% of 6,093) methods in ArgoUML and 1,434 (57.3% of 2,500) methods in jEdit. In total, the correct verbs are recommended in the top 10 for 60.6% of the methods covered by the rules. These results show that the naming association rules extracted from a set of software are effective in different projects.

The extracted rules covered 209 (76.8% of 272) verbs in ArgoUML and 217 (74.8% of 290) verbs in jEdit. This number is considerably larger than the 64 verbs covered by [6], [7]. We manually identified two groups of methods that no naming association rules could recommend the correct verbs for. One group of methods uses rare verbs used by a smaller number of methods than a threshold to extract rules. An example is `redo` that is likely a feature implemented by a few GUI applications. Some method names in the group could not be handled by OpenNLP, such as `put1`. The other group of methods is abstract methods. They do not have sufficient number of elements in their transactions.

B. RQ2: Are Naming Association Rules Meaningful?

We have manually analyzed what type of naming association rules recommended the correct verbs of methods in ArgoUML and jEdit. As a result, we have identified four groups of rules. Table II shows examples of the classified rules. The columns `Rule` and `Group` indicate identifiers for rules and identified groups. The columns `Antecedent`, `Consequent`, `Conf`, and `Sup`, respectively, indicate the antecedent, the consequent method-verb, the confidence, and the support of a rule.

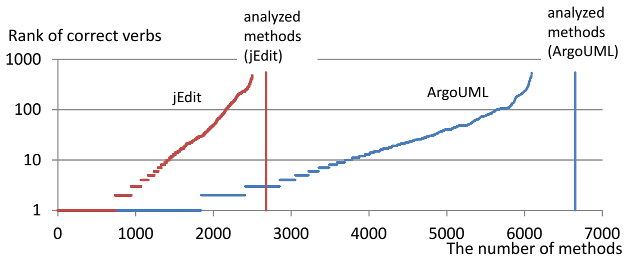

Fig. 2. Rank of correct verbs for methods in jEdit (left) and ArgoUML (right)

The first group of rules recommends the same verb as methods called in the method. R1 and R2 are example rules applied to methods in ArgoUML. R1 recommended `delete` for a method that calls `deleteInstance`. Similarly, R2 recommended `add` for a method that calls `addAll`. R8 is a rule applied to a method in jEdit. The rule also recommended `add` for a method that calls `add`. This group is consistent with a heuristic used by Sridhara *et al.*[15].

The second group recommends verbs that are conceptually related with a certain word in the method. R3 and R4 are examples in this group applied to methods in ArgoUML. Both rules recommended `execute` for methods related to `command`. R9 and R10 are rules applied in jEdit. The rules recommended `load` for methods related to `property`. This group might represent relationships between the verb and the direct object proposed by Shepherd *et al.* [16].

The third group recommends verbs related to a language specification. In this group, R5 recommends the `compare` method for the `Comparable` interface and R11 recommends the `run` method for the `Runnable` interface, respectively.

The fourth group recommends verbs based on Java programming idioms. R6 is a rule recommending `find` for methods using the `equals` method with `iterator`. This rule is similar to a typical behavior of the `find` methods identified in [6]. R12 recommended `copy` for methods using `read` and `write` for a stream.

While the four groups of rules captured meaningful rules, the verbs of several methods are recommended by less meaningful rules. For example, R7 recommended `reopen` for methods whose return type is `void`. This rule is applied to a method because it is a rule to recommend the correct verb for methods whose verb is `reopen`. Although at least 37 `reopen` methods are involved in the training dataset, there are no common identifiers among them except for `void`. Similarly, R13 represents that `use` methods are used for accessing the `boolean` flags in a program. As these rules have very low confidence values, a threshold for confidence could remove them from the result. However, determining an appropriate threshold is future work, because some meaningful rules (R6, R10 and R12) also have low confidence.

TABLE II
EXAMPLES OF RULES USED FOR METHODS IN ARGOUML AND JEDIT

Rule	Group	Software	Antecedent	Consequent	Conf	Sup
R1	1	ArgoUML	call:deleteInstance, return-type:void	delete	0.969305	600
R2	1	ArgoUML	call:addAll, return-type:boolean, call:size, argument-name:c	add	1.000000	22
R3	2	ArgoUML	argument-name:command, call:execute	execute	0.653061	32
R4	2	ArgoUML	parent-class-name:AbstractCommand	execute	0.765766	255
R5	3	ArgoUML	interface-name:Comparable, argument-type:Object, call:compareTo, return-type:int	compare	1.000000	281
R6	4	ArgoUML	call:next, call:hasNext, call:iterator, call:equals	find	0.065000	169
R7	-	ArgoUML	return-type:void	reopen	0.000111	37
R8	1	jEdit	argument-name:label, call:setConstraints, return-type:void, call:add	add	1.000000	27
R9	2	jEdit	return-type:Properties, call:load, call:getResourceAsStream	load	0.958333	23
R10	2	jEdit	call:getProperty, call:add	load	0.053571	27
R11	3	jEdit	call:error, return-type:void, interface-name:Runnable	run	0.626068	293
R12	4	jEdit	call:write, call:read, argument-type:InputStream, return-type:void	copy	0.216080	43
R13	-	jEdit	return-type:boolean	use	0.003684	411

V. THREATS TO VALIDITY

We extracted naming association rules from OSS projects. Although we collected them almost systematically, the result depended on the projects selected for the training dataset.

Some of the projects may include naming bugs. Since Høst *et al.* reported that naming bugs are found in at most 5% of the methods in a program, we believe that association rule mining does not extract such naming bugs as a rule recommending an inappropriate verb.

We have limited the number of elements in an antecedent in order to reduce the effort for a manual analysis of the rules. More complex but useful rules might be missing in our analysis because of the filtering conditions.

We used OpenNLP to split method names between a verb and objects. Since a programming language is not a natural language, transactions and rules may use incorrectly split identifiers. According to the manual analysis described in Section IV-B, we believe there are few such errors.

We applied the naming association rules to ArgoUML and jEdit. We selected them because both are famous OSS projects and often used for evaluation in other research studies. As a result, the target domain is limited to GUI applications. The application of the extracted rules to different domains may result in a different observation.

We have manually identified four groups of rules. The classification depends on the first author's experience. An expert of Java programming or OSS projects may identify different groups of rules.

VI. CONCLUSION

In this paper, we proposed a technique to recommend candidates of good method verbs to developers, using naming association rules. Our approach recommended correct verbs in the top 10 for 60.6% of the methods. Furthermore, the extracted rules covered 284 verbs and we could identify four meaningful groups of rules used for recommendation.

In the future work, we want to improve a ranking strategy to provide a better list of candidates to developers. An appropriate filter for rule mining is also important. We are interested in behavioral attributes such as nano-patterns [10] as additional clues to characterize the usage of verbs.

To achieve the full support for rename method refactoring, we need to recommend not only verbs but also objects in a method name. We are planning to take the domains of projects into account for recommendation, because different entities are frequently used in each domain.

Acknowledments: This work was supported by KAKENHI (Nos. 23680001, 25220003, and 25730036).

REFERENCES

[1] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?" in *Proc. ICPC*, 2012, pp. 193–202.

[2] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proc. ICPC*, 2006, pp. 3–12.

[3] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranič, "The emergent structure of development tasks," in *Proc. ECOOP*, 2005, pp. 33–48.

[4] S. McConnell, *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004.

[5] E. W. Høst and B. M. Østvold, "The Programmer's Lexicon, Volume I: The Verbs," in *Proc. SCAM*, 2007, pp. 193–202.

[6] ——, "Debugging Method Names," in *Proc. ECOOP*, 2009, pp. 294–317.

[7] E. K. Karlsen, E. W. Høst, and B. M. Østvold, "Finding and fixing Java naming bugs with the Lancelot Eclipse plugin," in *Proc. Workshop on Partial Evaluation and Program Manipulation*, 2012, pp. 35–38.

[8] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. International Conference on Management of Data*, 1993, pp. 207–216.

[9] Y. Hayase, Y. Kashima, Y. Manabe, and K. Inoue, "Building Domain Specific Dictionaries of Verb-Object Relation from Source Code," in *Proc. CSMR*, 2011, pp. 93–100.

[10] J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yiapanis, "Fundamental Nano-Patterns to Characterize and Classify Java Methods," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 191–204, 2010.

[11] "OpenNLP," http://opennlp.sourceforge.net/.

[12] "SourceForge," http://sourceforge.net/.

[13] "Apache Software," http://www.apache.org/.

[14] "Eclipse," http://www.eclipse.org/.

[15] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proc. ASE*, 2010, pp. 43–52.

[16] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Towards supporting on-demand virtual remodularization using program graphs," in *Proc. AOSD*, 2006, pp. 3–14.