IEICE TRANSACTIONS

on Information and Systems

VOL. E98-D NO. 3
MARCH 2015

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

Method Verb Recommendation Using Association Rule Mining in a Set of Existing Projects

Yuki KASHIWABARA[†a)], ***Nonmember***, **Takashi ISHIO**[†], ***Member***, **Hideaki HATA**[††], ***Nonmember***, ***and*** **Katsuro INOUE**[†], ***Fellow***

SUMMARY It is well-known that program readability is important for maintenance tasks. Method names are important identifiers for program readability because they are used for understanding the behavior of methods without reading a part of the program. Although developers can create a method name by arbitrarily choosing a verb and objects, the names are expected to represent the behavior consistently. However, it is not easy for developers to choose verbs and objects consistently since each developer may have a different notion of a suitable lexicon for method names. In this paper, we propose a technique to recommend candidate verbs for a method name so that developers can use various verbs consistently. We recommend candidate verbs likely to be used as a part of a method name, using association rules extracted from existing methods. To evaluate our technique, we have extracted rules from 445 open source projects written in Java and confirmed the accuracy of our approach by applying the extracted rules to several open source applications. As a result, we found that 84.9% of the considered methods in four projects are recommended the existing verb. Moreover, we found that 73.2% of the actual renamed methods in six projects are recommended the correct verb.

key words: *software readability, recommendation, method name, association rule*

1. Introduction

It is well-known that program readability is very important for maintenance tasks [1]. In software development, maintenance tasks occupy 80% of the software life cycle [2] and maintenance cost represents the true cost of software [3]. In performing the maintenance task, developers spend considerable time reading a program [4]. Consequently, high program readability leads to reduced maintenance cost.

An identifier is a crucial element for program readability [5]–[7]. The quality of identifiers affects program readability [5], [6]. Developers take a considerably longer time to understand a program if the identifiers poorly represent their roles in the program [6]. It is important to give easily recognized names to identifiers.

In identifiers, method names contribute to program readability because the names are used for understanding the behavior of the methods without reading the program. According to several guidelines for an object-oriented program [8]–[10], a method name generally consists of a verb and objects and should represent its behavior consistently throughout a program. Even if classes and methods are designed beforehand by an expert, developers may have to name private methods by themselves [11]. It is not easy for developers to choose verbs and objects consistently because each developer may have a different notion of a suitable lexicon for method names.

An inconsistent method name may create confusion for a developer. Figure 1 shows an example. In Java, a setter method is expected to update a field of an object. However, the code fragment gets a stream object from a blob object by calling the `setBinaryStream` method. The state of the blob object is updated by method calls to the stream object.

An integrated development environment IntelliJ* supports naming for well-known verbs that are already used consistently among developers. For example, it is well-known that `get` and `set` are used to represent accessor methods to read and update a field variable, respectively. The verb `test` is used to represent a unit test method for JUnit. If a method returns a boolean value, an asking verb or an auxiliary verb such as `is` or `can` will often be used for the method name. However, in the case of other verbs, it is not known when and which verbs should be used to name a method.

There are several techniques to recommend better verbs for a method name. In previous work, Karlsen *et al.* [12] implemented a naming bug detection tool based on naming rules extracted by their prior studies [3], [13]. Their tool accurately points out inappropriate verbs used for a method and recommends more appropriate verbs to the method. However, their tool is appropriate for naming bug detection only for limited methods by using a particular, narrow set of verbs. Shusi *et al.* proposed a technique to recommend a verb for a method name using machine learning [14]. Although their approach is applicable to many methods, and

```
...
FileInputStream is = new FileInputStream(file);
OutputStream os = blob.setBinaryStream(1);
c = is.read();
...
```

Fig. 1 A part of *t_createBlob_Client* method defined in *TestConnectionMethods* class in Derby 10.2.2.0.

Manuscript received August 11, 2014.
Manuscript revised November 25, 2014.
Manuscript publicized December 16, 2014.

[†]The authors are with the Osaka University, Suita-shi, 565–0871 Japan.
[††]The author is with the Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
a) E-mail: k-yuki@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2014EDP7276

*http://www.jetbrains.com/idea/

treats many verbs using a probability model, it does not explain why a verb is recommended. Only the probability value that a verb is most suitable is provided to a developer.

In this paper, we propose a technique to recommend candidate verbs for a method name so that developers can consistently use various verbs. We extract the relationship between verbs in method names and identifiers in methods from existing source files by using association rule mining [15]. We assume that the behavior of a method is often characterized by identifiers such as method calls and field access in the methods. Using the extracted rules, we recommend candidate verbs likely to be used as a part of a method name, along with the reason of recommendation: *e.g.*, *if a method calls* `next`, `hasNext`, `iterator`, *and* `equals`, *then* `find` *is likely to be a verb representing the behavior*. Our approach can be used after the implementation phase and before the maintenance phase to improve program readability.

We have extracted association rules from 445 open source projects written in Java and conducted two experiments to confirm the accuracy of our approach by applying three kinds of projects. The first experiments applied the rules to 1) the training data set and 2) four projects to confirm the accuracy of rule extraction. The second round of experiments applied rules to actual renamed methods in 3) six projects to confirm the accuracy of verb recommendation. As a result, 1) naming association rules were extracted from 84.5% of methods in training data set, and covered 230 verbs. 2) We found that 84.9% of the considered methods in four projects are recommended the existing verb. We have identified three meaningful groups of rules for verb recommendation. 3) We found that 72.4% of the renamed methods in six projects are recommended the correct verb and 51.5% of the renamed methods are recommended the correct verb at a higher rank than the verb prior to the change.

The main contributions of this paper are as follows:

- We have defined an application of association rule mining to extract relationships between verbs used in method names and identifiers in methods.
- We have shown that association rules extracted from open source projects are applicable to the recommendation of candidate verbs for methods in different applications.
- We have shown that the technique could recommend correct verbs for 72.4% of the existing methods covered by our approach.

This paper is a revised version of [16]. This paper includes an additional experiment that compares the result of our technique with actual methods renamed by developers. In addition, we have updated our tool with refined heuristics. Therefore, the result reported in this paper is different from that report.

The rest of this paper is organized as follows: Sect. 2 explains the related works of our research. Section 3 describes our approach to recommending candidates verbs. Section 4 shows the result of our experiment. Section 5 discusses threats to the validity of the proposed approach and the experiment. Section 6 presents the conclusions and future work.

2. Related Work

2.1 Rename Refactoring

Refactoring is used to improve software quality [17]. Refactoring is defined as *the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure* [18]. Refactoring is one of the most important and commonly used techniques for improving software quality [19].

There are several tools used to automatically execute refactoring. For example, IntelliJ and Eclipse contain the function as plug-ins. Murphy-Hill *et al.* researched the refactoring behavior of programmers in [20]. Their data indicated that developers actually used tools for automatic refactoring. Bavota *et al.* concluded that the refactoring has less to do with the percentage of faults [21]. Bavota *et al.* suggested that developers should conduct automatic refactoring using tools rather than manual refactoring [21].

Opdyke *et al.* reported that renaming is one of the most commonly used refactorings [22]. Arnaoudova *et al.* [23] noted that 39% of 71 developers of industrial or open source systems perform refactoring from a few times per week to almost every day and 46% perform refactoring only a few times per month. Moreover, they analyzed identifier renaming across versions of a program and showed that method names are renamed more frequently than other identifiers. Consequently, it is important to support method renaming.

2.2 Method Naming

Both Abebe *et al.* and Arnaoudova *et al.* defined bad patterns of identifiers and implemented detectors for those patterns. Abebe *et al.* defined "lexicon bad smells" for identifier naming [24]. Lexicon bad smells are potential structural problems of identifiers caused by the naming itself or the relation among identifiers. Arnaoudova *et al.* defined anti-patterns for identifier naming [25]. The anti-patterns are represented by the relationships among the signature of a method, identifiers involved in the method body, and the comment of the method. Neither research group considered an appropriate and concrete lexicon for identifier names.

Høst *et al.* analyzed the relationship between the behavior of methods and the verbs used in the method names [13]. For each verb, they analyzed the typical behavior of methods including the verb in their names. They defined traceable attributes such as "create objects" and "contains loop" to represent the behavior of methods. Finally, they reported 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 that highlights the naming bugs of methods to developers and that provides suggestions as to how to fix the naming bugs [3]. The technique is implemented as a tool by Karsen *et al.* [12]. This tool points out that a target method has a naming bug, if the method's name has very little correspondence with the rule. Although the technique is highly accurate for naming bug detection, developers can receive support for only a limited number of methods whose names are covered by 76 method name patterns using 64 verbs. Each method name pattern is associated with a single usage rule. The technique is hard to cover various verbs, because a verb may be used in many ways according to the local definition or idiosyncrasies [24]. In our work, we use association rule mining to cover a large number of verbs and to apply the approach to many methods. Our approach help developers to consistently use verbs in addition to changing inappropriate verbs used in method names.

Shusi *et al.* proposed a technique to recommend a verb for a method name using a machine learning technique [14]. They classified methods using support vector machine. The classifier regards each verb of a method name as a label of a clustering. Although it can be applied to various methods, developers still have to verify the resultant verb by themselves. We use a rule mining approach to provide rules between method contents and names so that developers can understand why a verb is recommended.

Suzuki *et al.* proposed a technique to recommend method objects using an n-gram model [26]. They recommended a word that is likely to be added to the end of a method name, using the conditional probability extracted from existing method names. Their approach cannot recommend the first word for a method, whereas our technique can. We consider that we may recommend whole method names in combination with their approach.

3. Proposed Approach

We recommend candidate verbs for a method name using association rule mining. The proposed approach consists of two steps. The first step extracts naming association rules from verbs used in method names and the identifiers in methods. The second step applies the rules to recommend verbs for a method name. We call the extracted rules **naming association rules**.

3.1 Extraction of Naming Association Rules

In this step, our approach extracts naming association rules from a training data set using association rule mining. Association rule mining [15] is a technique used for extracting association rules from a large number of sets of items. A set of items is called a *transaction*. An association rule can be denoted as (X, Y, c, s). This expression represents that, if a transaction contains all items in X, then the transaction likely contains items in Y. X is called the antecedent of the rule, Y is called the consequent of the rule. c is a confidence value, which is the conditional probability of Y given X; s is a support value, which is the number of transactions that fulfill the rule. In this paper, a training data set represents a set of existing projects. To that end, our approach takes methods from a training data set, translates methods to transactions, and extracts naming association rules from transactions.

First, our approach takes methods M from source files in the training data set. We create an AST-tree for each source file and extract M from the source files. We split method names by using camelCase and snake_case heuristics like [25]. We exclude the following methods from M.

abstract methods: These methods have very few information for recommendation.

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

methods defined in anonymous inner class: Most of these 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`.

methods not starting with a verb as judged by OpenNLP: We extract rules from methods where there is no uncertainly as to whether the word in question is used as a verb. OpenNLP[†] is a natural language processing tool. We use WordNet[††] as the library. 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.

methods not starting with an uncapitalized word: Generally, method names should start with an uncapitalized verb, so we exclude methods like `OpenInputfile`.

Second, our approach generates 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 six types of elements in a method m defined in C as $t(m)$.

method-verb: A verb used in the name of m. We do not use stemming. We analyze similar verbs including synonyms individually.

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.

[†]http://opennlp.sourceforge.net/
[††]http://wordnet.princeton.edu/

call: Name of a method directly called by m.

We regard several reserved words used for types like `boolean` and `void` as identifiers. 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."

Figure 2 shows how a source file is translated into transactions. The source file in Fig. 2 (a) includes two methods: `findName` and `addName`. Figures 2 (b) and (c) represent t(`findName`) and t(`addName`), respectively.

Finally, our approach applies association rule mining to the transaction set T. We have established two conditions for extraction of naming association 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 v denotes the consequent {method-verb:v}. For example, out of 100 methods whose verb is `add`, if 80 of them have an `addAll` method in their method definitions, a naming association rule ({call:addAll}, add, 0.8, 80) is extracted.

Furthermore, we use the following two conditions for further filtering.

3) The number of items in antecedent $|X| \leq 4$.
4) Support $s \geq 100$.

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. Høst *et al.* also defined the heuristic threshold which methods must cover in at least 100 methods for rule detection [3].

```
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 != null){ nameList.add(index, name); }
 }
}
```

(a) a source file

method-verb:	find
return-type:	String
argument-type:	String
argument-name:	name
field-name:	nameList
call:	contains

(b) the transaction extracted from `findName` method

method-verb:	add
return-type:	void
argument-type:	int
argument-name:	index
argument-type:	String
argument-name:	name
field-name:	nameList
call:	add

(c) the transaction extracted from `addName` method

Fig. 2 An example of a translation from a Java source file.

3.2 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.

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

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. A list of verbs sorted by descending order of their confidence values is recommended to developers.

4. Evaluation

We have evaluated the accuracy of our approach from two points of view, each of which contains two research questions:

1) Rule extraction

RQ1 How many existing verbs are covered?
RQ2 What kinds of rules are used for recommendation?

2) Verb recommendation

RQ3 How many verbs can be recommended correctly?
RQ4 How many correct verbs can be recommended at a higher rank than those used before the change?

For evaluation, we extracted naming association rules from a training data set: 445 open source projects which are written in Java and which are obtained from sourceforge.net[†], apache.org[††], and eclipse.org[†††]. The number of the extracted naming association rules is 82,102.

We applied the extracted rules to the training data set itself and four open source projects: ArgoUML, Berkeley, Castor, and Order Portal. We applied the extracted rules to renamed methods extracted from six repositories of open source projects: eclipse.pde.ui, org.eclipse.efc, org.eclipse.mwe, cassandra, jmeter, and axis1-java. The training data set does not include those ten projects.

Tables 1 and 2 present an overview of the training data set and the target projects with the results, respectively. Domain represents the type of the domain of a project. We referred to [27] with regard to classification. #LOC denotes the number of lines in the source files. #File indicates the number of files in the project, and #Target represents the number of considered methods as described in Sect. 3.1. #Recomm denotes the number of methods whose existing

[†]http://sourceforge.net/
[††]http://www.apache.org/
[†††]http://www.eclipse.org/

Table 1 Overview of training data set and target projects.

Target Project	Domain	#LOC	#File	#Target	#Recomm	Top10	#Verb
Training data set	(many)	34,326,308	196,947	361,093	305,068(84.5%)	192,857(63.2%)	230/2464(0.1%)
ArgoUML 0.28.1†	GUIs	367,052	1,994	3,712	3,072(82.6%)	1,629(53.0%)	138/252(54.8%)
Berkeley DB Java Edition 4.0.92††	Databases	181,198	1,198	3,238	2,633(81.3%)	1,324(50.3%)	136/270(50.4%)
Castor 1.3†††	XML	306,376	1,735	2,422	2,146(88.6%)	1,477(68.8%)	115/201(57.2%)
Order Portal 1.2.4††††	Web Applications	473,826	709	2,505	2,231(89.1%)	821(36.7%)	95/160(59.4%)
Sum for four projects	(four domains)	1,328,452	5,636	11,877	10,082(84.9%)	5,251(52.1%)	192/505(38.0%)

Table 2 Overview of target projects.

Target Project	Period	#Rename	#Recomm	Top10
org.eclipse.mwe†††††	09/20/2007 –11/25/2008	7	5	3
org.eclipse.ecf††††††	12/03/2004 –02/21/2014	15	11	3
eclipse.pde.ui*	05/24/2001 –02/18/2014	7	5	2
Cassandra**	03/02/2009 –09/13/2013	35	20	8
axis1-java***	09/03/1998 –11/01/2011	20	20	6
jmeter****	01/19/2001 –12/10/2012	13	10	2
sum	(NONE)	97	71(73.2%)	24

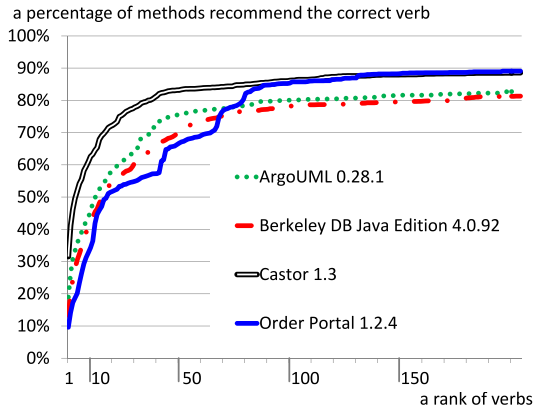

Fig. 3 Rank of correct verbs for methods in four projects.

verbs are recommended in each list (and the percentage of methods to target methods). Top10 indicates the number of methods recommended in the top 10 of each list (and the percentage). #Verb represents the number of different kinds of verbs recommended by extracted rules against(/) the number of all kinds of verbs in the project (and the percentage). Period denotes the development period of the each project. #Rename represents the number of renamed methods extracted automatically from the six projects and validated manually by the first author.

4.1 How Many Existing Verbs are Covered?

To answer this research question, we have evaluated whether existing verbs used in a program can be recommended by the extracted rules. We applied our approach to methods in the training data set and in four projects from different domains. We have computed the rank of the existing verb in the recommendation list for each method.

Table 1 represents the result of recommendation for the training data set and other four projects. For the training data set, the existing verbs are recommended for 84.5% of methods. It implies that the naming rules are extracted from 84.5% methods in the training data set. Our approach recommended the existing verb in the top 10 of a ranking for 63.2% methods covered by extracted rules.

Figure 3 shows a plot of the results of recommendation for four projects. The vertical axis represents the percentage of the number of methods recommended for the existing verb. The horizontal axis represents the rank of verbs. From Fig. 3, if we check the top ten of the recommended list of ArgoUML, we can find existing verbs for 44.9% of target methods.

For other projects, the existing verbs are recommended for 84.9% of the methods, on average. We have found that our approach can recommend verbs to as many methods exist in the training data set, regardless of the domain. With regards to methods recommended in top 10, the existing verbs are recommended for 52.1% of the methods covered by our approach on average; 68.8% of Castor is higher and 36.7% of Order Portal is lower than the average of the four projects. We consider that the percentage of methods for which existing verbs are recommended has little variation within each domain; however, the percentage of methods for which existing verbs are recommended in the top 10 does significantly vary in each domain. These results show that the naming association rules extracted from a set of software are effective for projects in different domains; however, the ranking strategy should be changed to suit the relevant domain.

The extracted rules can recommend 230 verbs and covered 192 verbs in four projects. This number is considerably larger than the 64 verbs covered by [3]. This number is also practically as large as the 237 verbs covered by [14].

†http://argouml.tigris.org/
††http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html
†††http://castor.codehaus.org/index.html
††††http://www.randrinc.com/2d_randr/orderportal/home
†††††http://git.eclipse.org/gitroot/emf/org.eclipse.mwe.git
††††††http://git.eclipse.org/gitroot/ecf/org.eclipse.ecf.git
*http://git.eclipse.org/gitroot/pde/eclipse.pde.ui.git
**git://git.apache.org/cassandra.git
***git://git.apache.org/axis1-java.git
****git://git.apache.org/jmeter.git

We manually identified two groups of methods for which the correct verbs were not recommended. First, most of such methods use rare verbs utilized by a smaller number of methods than our support threshold of 100. Some verbs may be recommended when we lower the threshold for rule extraction. For example, there are 85 methods using the verb `warning` in the training set. Second, in some methods, the name can not be split by camelCase or an under_score, such as `addto`. If we use a novel identifier splitter like [28], we can identify such verbs as `addto` as those containing the verb `add`.

4.2 What Kinds of Rules are Used for Recommendation?

To understand why the approach recommends certain verbs, we have manually analyzed what type of naming association rules recommended the existing verbs of methods in ArgoUML. We analyzed 347 rules used to recommend existing verbs for more than two methods in the ArgoUML project.

As a result, we have found three meaningful groups. Table 3 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` indicate the antecedent, the consequent method-verb, the confidence, and the support of a rule, respectively.

The first meaningful group of rules recommends the same verb as methods called within the method. In this group, R1 recommends `remove` for a method that calls `remove`. Similarly, R2 recommends `generate` for a method that calls `generate`. This group is consistent with a heuristic used by Sridhara *et al.* [29].

The second group recommends verbs that are conceptually related with a certain word in the method. In this group, R3 recommends a verb `init` to methods that call `setText` and `setEditable` methods. This rule implies that, in some cases, methods whose verb is `set` set some properties to initialize some objects. Methods whose verb is `init` may call many `set` methods. R4 recommends `parse` for methods that call a `nextToken` method, and R5 recommends `parse` for methods that call `nextToken` and `hasMoreTokens` methods. We suggest that the rules recommended `parse` for methods related to `token` because there are more than two rules whose antecedents have an identifier including the word `token`. Some rules such as R4 and R5 might represent relationships between the verb and the direct object as proposed by Shepherd *et al.* [30].

The third group recommends verbs based on Java programming idioms. In this group, both R6 and R7 recommend `is` for methods returning the `boolean` value. These rules represent a naming convention implemented in IntelliJ.

Although the three groups of rules captured meaningful rules, the verbs of several methods are presented by less meaningful rules. For example, R8 represents that `end` methods return the `void`: there are no common identifiers among them except for `void`. R9 represents that `accept` methods return the `boolean` and use a `File` object, whereas various methods use files and boolean flags. The verbs are hard to predict from the antecedent of the rules. Consequently, finding the appropriate threshold remains a target for future work.

4.3 How Many Verbs Can Be Recommended Correctly?

We investigated whether our approach can correctly recommend verbs for actual methods renamed by developers. We assume that developers should have renamed methods appropriately

We extracted renamed methods from repositories of target projects using a technique proposed by Hata *et al.* [31]. A **renamed method** is a method whose name is different between the two adjacent revisions. The method in the previous revision is called the **original version** of the renamed method. The method in the newer revision is called the **changed version** of the renamed method. We call the verbs used in each version of a renamed method the **original verb** and **correct verb**, respectively.

We made following four conditions for extraction of renamed methods. We set up the condition 1), in order to be able to compare the rank between a correct verb and an original verb. We set up the conditions 2), 3), and 4), in order to be able to validate the methods manually.

1) an original verb and a correct verb are different, respectively.
2) an original version and a changed version of a renamed method have more than five lines of code.
3) a renamed method is defined in the same directory and class in each revision.
4) the similarity between the original version and the

Table 3 Examples of rules used for methods in ArgoUML.

Rule	Group	Antecedent(X)	Consequent(v)	Conf(c)	Sup(s)
R1	Meaningful 1	call:remove, argument-type:Object	remove	0.491018	328
R2	Meaningful 1	call:generate	generate	0.581818	128
R3	Meaningful 2	call:setText, call:setEditable, return:void	init	0.488281	125
R4	Meaningful 2	call:indexOf, call:length, call:nextToken, argument-type:String	parse	0.559140	104
R5	Meaningful 2	argument-type:String, call:hasMoreTokens, call:equals, call:nextToken	parse	0.431877	168
R6	Meaningful 3	call:booleanValue, return:boolean	is	0.848871	1,466
R7	Meaningful 3	call:startsWith, return:boolean	is	0.395299	370
R8	less meaningful	return:void	end	0.011185	2,297
R9	less meaningful	argument-type:File, return:boolean	accept	0.201811	156

changed version of a renamed method, which is computed by Git, is more than 50%.

We extracted renamed methods from six projects, and validated them manually. If a verb in a method name is renamed more than once, we remove the old versions of the method and select only the newest two versions as a target renamed method. We removed a renamed method if the renamed method extracted automatically has no correspondence between the two version. Then, we extracted 97 renamed methods in total.

We have applied our approach to renamed methods and computed the rank of the correct verb in the list for each renamed method. Applying our approach to a renamed method means that we apply our approach to the changed version of the renamed method and get recommendation list for the changed version.

Table 2 represents the result. As a result, 73.2% of renamed methods are recommended the correct verb. The proposed approach recommended the correct verb in the top 10 of a ranking for 33.8% of renamed methods covered by our approach. We found that our approach recommended correct verbs to as many renamed methods as exist in the training data set. We believe that naming association rules extracted from the training data set can recommend correct verbs sufficiently. In the rules used for the recommendation, there are some that can be classified as meaningful rules explained in Sect. 4.2, which recommend the same verb for methods called in the method. For example, one rule recommended `remove` to methods that call `remove` methods, as shown in Fig. 4.

4.4 How Many Correct Verbs can Be recommended at a Higher Rank than the Original Verbs?

We investigated whether there is the potential for developers to use our approach effectively. We compared the rank between a correct verb and an original verb in each list for a renamed method. Our approach is effective in renaming actual methods if the rank of a correct verb is higher than an original verb.

Table 4 represents the results using a pivot table. We categorized the rank of verbs into three categories: 1) in top 10, 2) out of top 10, and 3) out of ranking. From Table 4, there are nine method pairs where the original verb is recommended out of the top 10 and the correct verb is recommended in the top 10. We found that there are 50 renamed methods whose correct verbs are recommended at a higher rank than the original verbs (orange cells). On the other hand, 34 renamed methods are otherwise (blue cells). We found that there are more renamed methods whose correct verbs are recommended at a higher rank than the original verbs.

Moreover, we have manually analyzed what kinds of renaming occurred in each renamed method, comparing two Java files in which a renamed method is defined. There is no renamed method where the correct verb is not clearly more appropriate verb than the original verb.

Figure 4 represents one of the examples whose correct verb is recommended at a higher rank than the original verb. The method's verb of the method is renamed from `get` to `remove`. Although `get` is not recommended in the list, `remove` is recommended at the top of the list. It is thought that `get` is not suitable to the method because the verb `get` is used to represent an accessor. We consider it inconsequential that our approach cannot recommend `get` to any method because `get` is removed from rule extraction in advance. We consider that it to be a good recommendation from identifiers in the method definition using our approach.

Figure 5 represents another example where the correct verb is not recommended at a higher rank than the original verb. A verb of the method is renamed from `close` to `stop`. `close` is recommended at the top of the list, but `stop` is recommended out of the top 10 (top 23) in the list. This target method is a private method called by a `stop` method, which is defined in BundleActivator interface. This `stop`

Table 4 The rank of correct verbs for methods renamed by developers.

origin verbs \ correct verbs	in top 10		out of top 10		out of the list	sum
in top 10	higher: 3	lower: 5	12		5	25
out of top 10	9		higher: 9	lower: 4	8	30
out of the list	9		20		13	42
sum	26		45		26	97

```
public DiscoveredEndpointDescription
        removeDiscoveredEndpointDescription(
         IDiscoveryLocator locator, IServiceID serviceID){
 synchronized (discoveredEndpointDescriptions) {
   DiscoveredEndpointDescription ded
   = findUniscoveredEndpointDescription(locator, serviceID);
    if (ded != null) {
      // remove
      discoveredEndpointDescriptions.remove(ded);
      return ded;
    }
 }
 return null;
}
```

Fig. 4 A method defined in *DiscoveredEndpointDescriptionFactory* class in org.eclipse.ecf.

```
private void stopEndpointDescriptionFactory(){
 if(endpointDescriptionFactoryRegistration != null){
  endpointDescriptionFactoryRegistration.unregister();
  endpointDescriptionFactoryRegistration = null;
 }
 if (endpointDescriptionFactory != null) {
  endpointDescriptionFactory.close();
  endpointDescriptionFactory = null;
}
```

Fig. 5 A method defined in *Activator* class in org.eclipse.ecf.

```
public void receive(MessageContext messgeCtx)throws AxisFault{
 RelatesTo relatesTO =messgeCtx.getMessageInformationHeaders()
             .getRelatesTo();

 String messageID = relatesTO.getValue();
 Callback callback = (Callback) callbackstore.get(messageID);
 AsyncResult result = new AsyncResult(messgeCtx);
 if (callback != null) {
  callback.onComplete(result);
  callback.setComplete(true);
 } else {
   throw new AxisFault(
    "The Callback realtes to MessageID " + messageID +
                                        " is not found");
 }
}
```

Fig. 6 A method defined in *CallbackReceiver* class in axis1-java.

method calls eight private methods whose names start with `close`, which are renamed from `close` to `stop`, as in this example. In analyzed renamed methods, six renamed methods including this example are related to this change. We consider that developers renamed the methods according to the caller's name rather than their code. Our approach could not work effectively because such a cause of renaming is not taken into account.

Figure 6 represents an example whose correct verb is recommended at a higher rank than the original verb, but our approach still did not work effectively. The method is replaced the misspelled word `recieve` with `receive`. Although `recieve` is not recommended in the list, `receive` is recommended out of the top 10 (top 133) in the list. This method is defined in the parent class as an abstract method. Our method renaming was affected by the renaming in the parent class. In analyzed renamed methods, four renamed methods including this example are related to this change. So, even if we recommend verb candidates to the method from the body, the name may not be legitimately changed. Clearly, our approach does still have considerable potential for improvement.

As a whole, the number of renamed methods whose correct verbs are recommended at a higher rank than the original verbs is almost the same as the number of renamed methods which are not so. We consider that our approach has potential for improvement if using additional information is utilized.

5. Threats to Validity

We extracted naming association rules from open source projects. Although we collected them almost systematically, the result is dependent on the projects selected for the training data set.

Methods in the training data set may include naming bugs. Because Høst *et al.* reported that naming bugs are found in at most 5% of the methods in a project, we believe that association rule mining does not extract many rules recommending an inappropriate verb.

We have limited the number of elements in an antecedent in order to reduce the effort required 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 and WordNet to check whether the words are verbs or not. Because a programming language is not a natural language, methods may use different lexicons for verbs. Arnaoudova *et al.* also used WordNet, and said that WordNet is not optimal for programming lexicons, but it can be replaced easily [25].

In the experiment, we assume that existing verbs are appropriate for each method. Although this assumption has no solid evidence, a previous work Shusi *et al.* [14] also used the same assumption.

We did not evaluate the precision and the recall, because we regarded the existing verb in a method as only one basis of the evaluation. Shusi *et al.* indicated only the accuracy of their approach [14]. We believe that our approach was evaluated sufficiently in regards only to the accuracy.

We have manually identified three groups of rules. The classification depends on the first author's experience. If other experts classified them, the results may be different.

We have manually analyzed renamed methods. The judgment of why certain methods are renamed depends on the first author's opinion. If experts of each open source project were to carry out the some judgment, the results may be different.

We considered only methods whose names start with a verb. A guideline suggests that a verb should be used for methods writing some attributes and that a verb should not be used for methods only reading some attributes [10]. In this paper, our target is verb recommendation for methods whose names start with a verb. We believe that this has little impact because we remove methods whose names do not start with verbs for rule extraction and for evaluation.

6. Conclusion

In this paper, we proposed a technique to recommend candidates of appropriate method verbs to developers, using naming association rules. The extracted rules covered 230 verbs and recommended verbs for 84.9% of methods in four projects not included in the target projects. Our approach recommended better verbs for 73.2% of the renamed methods extracted from six projects. Furthermore, we identified three meaningful groups of rules used for recommendation.

In future work, we would like to reconsider thresholds for association rule mining and improve a ranking strategy to provide a better list of candidates to developers. We are also interested in additional information to characterize the usage of verbs for rule mining, e.g., nano-patterns [32], calling context of methods as discussed in Sect. 4.4, and domain information of analyzed software. To achieve full support for rename refactoring, we may recommend whole method names in combination with [26]. Finally, we would like to evaluate the effectiveness of our approach for software maintenance with human subjects.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers 25220003, 26280021, and 26540029.

References

[1] E. Collar and R. Valerdi, "Role of software readability on software development cost," Proc. COCOMO/SCM, 2006.
[2] F. Corbo, C. Del Grosso, and M. Di Penta, "Smart formatter: Learning coding style from existing source code," Proc. ICSM, pp.525–526, 2007.
[3] E.W. Høst and B.M. Østvold, "Debugging method names," Proc. ECOOP, pp.294–317, 2009.
[4] G.C. Murphy, M. Kersten, and M.P. Robillard, "The emergent structure of development tasks," Proc. ECOOP, pp.33–48, 2005.
[5] F. Deissenboeck and M. Pizka, "Concise and consistent naming," Software Quality Control, vol.14, no.3, pp.261–282, 2006.
[6] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," Proc. ICPC, pp.3–12, 2006.
[7] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?," Proc. ICPC, pp.193–202, 2012.
[8] S. McConnell, Code Complete, Second Edition, Microsoft Press, 2004.
[9] "Java Code Conventions," 1997.
[10] R. Green and H. Ledgard, "Coding guidelines: Finding the art in the science," Commun. ACM, vol.54, no.12, pp.57–63, 2011.
[11] A. Thies and C. Roth, "Recommending rename refactorings," Proc. RSSE, pp.1–5, 2010.
[12] E.K. Karlsen, E.W. Høst, and B.M. Østvold, "Finding and fixing Java naming bugs with the Lancelot Eclipse plugin," Proc. PEPM, pp.35–38, 2012.
[13] E.W. Høst and B.M. Østvold, "The Programmer's lexicon, Volume I: The verbs," Proc. SCAM, pp.193–202, 2007.
[14] S. Yu, R. Zhang, and J. Guan, Properly and Automatically Naming Java Methods: A Machine Learning Based Approach, pp.235–246, Springer Berlin Heidelberg, 2012.
[15] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," Proc. SIGMOD, pp.207–216, 1993.
[16] Y. Kashiwabara, Y. Onizuka, T. Ishio, Y. Hayase, T. Yamamoto, and K. Inoue, "Recommending verbs for rename method using association rule mining," Proc. CSMR-WCRE, pp.323–327, 2014.
[17] T. Mens and T. Tourwe, "A survey of software refactoring," TSE, vol.30, no.2, pp.126–139, 2004.
[18] W.F. Opdyke, Refactoring Object-oriented Frameworks, Ph.D. thesis, 1992.
[19] K. Stroggylos and D. Spinellis, "Refactoring–Does it improve software quality?," Proc. WoSQ, pp.10–15, 2007.
[20] E. Murphy-Hill, C. Parnin, and A.P. Black, "How we refactor, and how we know it," TSE, vol.38, no.1, pp.5–18, 2012.
[21] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? An empirical study," Proc. SCAM, pp.104–113, 2012.
[22] W.F. Opdyke and R.E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," SOOPPA, pp.145–161, 1990.
[23] V. Arnaoudova, L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.G. Gueheneuc, "REPENT: Analyzing the nature of identifier renamings," TSE, vol.40, no.5, pp.502–532, 2014.
[24] S.L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," Proc. WCRE, pp.95–99, 2009.
[25] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.G. Gueheneuc, "A new family of software anti-patterns: Linguistic anti-patterns," Proc. CSMR, pp.187–196, 2013.
[26] T. Suzuki, K. Sakamoto, F. Ishikawa, and S. Honiden, "An approach for evaluating and suggesting method names using N-gram models," Proc. ICPC, pp.271–274, 2014.
[27] Y. Hayase, Y. Kashima, Y. Manabe, and K. Inoue, "Building domain specific dictionaries of verb-object relation from source code," Proc. CSMR, pp.93–100, 2011.
[28] A. Corazza, S.D. Martino, and V. Maggio, "LINSEN: An efficient approach to split identifiers and expand abbreviations," Proc. ICSM, pp.233–242, 2012.
[29] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," Proc. ASE, pp.43–52, 2010.
[30] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Towards supporting on-demand virtual remodularization using program graphs," Proc. AOSD, pp.3–14, 2006.
[31] H. Hata, O. Mizuno, and T. Kikuno, "Historage: Fine-grained version control system for Java," Proc. IWPSE-EVOL, pp.96–100, 2011.
[32] J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yiapanis, "Fundamental nano-patterns to characterize and classify Java methods," ENTCS, vol.253, no.7, pp.191–204, 2010.

Yuki Kashiwabara received her B.E. from Osaka University in 2012. She is a Master candidate at Osaka University since 2013. Her research interests include software readability, and rename refactoring. She is a member of the IEEE and the IEEE Computer Society.

Takashi Ishio received the Ph.D. degree in information science and technology from Osaka University in 2006. He was a JSPS Research Fellow from 2006–2007. He is now an assistant professor of computer science at Osaka University. His research interests include program analysis and program comprehension. He is a member of the IEICE, IPSJ, JSSST, IEEE, and ACM.

Hideaki Hata received the Ph.D. degree in information science and technology from Osaka University in 2012. He was a JSPS Research Fellow from 2011–2013. He is now an assistant professor at Nara Institute of Science and Technology. His research interests includes software analytics, software economics, and human software interaction. He is a member of the IPSJ, JSSST, IEEE, IEEE Computer Society, and ACM.

Katsuro Inoue received the B.E., M.E., and D.E. degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984–1986. He was a research associate at Osaka University from 1984–1989, an assistant professor from 1989–1995, and a professor beginning in 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.