Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Toward an accurate method renaming approach via structural and lexical analyses*

Junpeng LUO[1], Jingxuan ZHANG[‡1,2], Zhiqiu HUANG[1], Yong XU[3], Chenxing SUN[3]

[1]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China*

[2]*Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 211106, China*

[3]*Tencent Technology (Shenzhen) Company Limited, Shenzhen 518054, China*

E-mail: luojunpeng@nuaa.edu.cn; jxzhang@nuaa.edu.cn; zqhuang@nuaa.edu.cn; rogerxu@tencent.com; marssun@tencent.com

Received Sept. 30, 2021; Revision accepted Feb. 28, 2022; Crosschecked Mar. 24, 2022

Abstract: Methods in programs must be accurately named to facilitate source code analysis and comprehension. With the evolution of software, method names may be inconsistent with their implemented method bodies, leading to inaccurate or buggy method names. Debugging method names remains an important topic in the literature. Although researchers have proposed several approaches to suggest accurate method names once the method bodies have been modified, two main drawbacks remain to be solved: there is no analysis of method name structure, and the programming context information is not captured efficiently. To resolve these drawbacks and suggest more accurate method names, we propose a novel automated approach based on the analysis of the method name structure and lexical analysis with the programming context information. Our approach first leverages deep feature representation to embed method names and method bodies in vectors. Then, it obtains useful verb-tokens from a large method corpus through structural analysis and noun-tokens from method bodies through lexical analysis. Finally, our approach dynamically combines these tokens to form and recommend high-quality and project-specific method names. Experimental results over 2111 Java testing methods show that the proposed approach can achieve a Hit Ratio, or Hit@5, of 33.62% and outperform the state-of-the-art approach by 14.12% in suggesting accurate method names. We also demonstrate the effectiveness of structural and lexical analyses in our approach.

Key words: Method renaming; Code refactor; Deep learning; Convolutional neural networks
https://doi.org/10.1631/FITEE.2100470 **CLC number:** TP311

1 Introduction

Method names are extremely important when developers program, as Høst and Østvold (2009) strongly noted: "methods are the smallest named units of aggregated behavior in most conventional programming languages and hence the cornerstone of abstraction." However, giving methods appropriate names is one of the most difficult tasks for developers (Allamanis et al., 2015), because it is not easy for developers to select suitable constitutive terms for identifiers while following the corresponding code conventions. Hence, it is common to find many flawed method names, such as non-standard method names or method names that are inconsistent with their method bodies, in real software projects.

Fig. 1 shows three examples of flawed method

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (Nos. 61902181 and 62002161), the China Postdoctoral Science Foundation (No. 2020M671489), the CCF-Tencent Open Research Fund (No. RAGR20200106), and the Nanjing University of Aeronautics and Astronautics Postgraduate Research and Practice Innovation Program (No. xcxjh20211612)

ORCID: Jingxuan ZHANG, https://orcid.org/0000-0002-8437-6640

© Zhejiang University Press 2022

names from a real software project, i.e., Cassandra of Apache (https://github.com/apache/cassandra/). As shown in Fig. 1, the name of the first method is a non-standard name with typos, in which diffrence should be difference. From the return type of the second method, we know that it should return a Boolean value. However, the current method name does not have a verb-token to represent its function. According to Java programming specifications, methods that return Boolean values should have the prefix "is," so a better method name may be isIncrementalBackups. The third method name also needs improvement, because it does not specify the granularity of column index size, e.g., B, KB, MB, or others. In contrast, its method body specifies the exact granularity. Hence, a better name for this method may be getColumnInedxSizeInKB.

```
public long diffrence(){
      long current = meter.count();
      long difference = current - reported;
      this.reported = current;
      return difference; }

public static boolean incrementalBackups(){
      return conf.incrementalbackups; }

public static int getColumnIndexSize(){
      return columnIndexSizeInKB; }
```

Fig. 1 Three real examples of flawed method names from Apache

After demonstrating some actual flawed method names, we continue to investigate their potential impacts in some open technical forums and platforms. Specifically, we first define a series of search queries by combining the compound conjunction "method name" with some adjective keywords, e.g., "inconsistent," "non-standard," and "rename." Then, we input these search queries in the search engines into Stack Overflow (https://stackoverflow.com/) and GitHub (https://github.com/), respectively, to match relevant posts and commit logs. As a result, 3571 posts in Stack Overflow and 1 337 519 commits in GitHub were returned. This means that there is a lot of concern from developers about non-standard and inconsistent method names. Fig. 2 shows some examples of retrieved results. As a proposed question in Stack Overflow shown on the left in Fig. 2, one developer requests a refactor safe way to get the name of a method, which reflects that inaccurate method names may lead to software security problems. As the commits in GitHub shown on the right in Fig. 2, we can see that flawed method names have already resulted in some bugs, and that developers pay much attention to renaming methods to generate more understandable names.

It is generally believed that accurate method names make methods more understandable and maintainable, and that inaccurate method names make methods difficult to understand and maintain (Takang et al., 1996; Lawrie et al., 2006; Arnaoudova et al., 2014; White et al., 2016), and may even lead to software defects (Butler et al., 2009; Abebe et al., 2011, 2012; Amann et al., 2019). Hence, it is vital to rename these inaccurately named methods in the practical software development process.

Existing similar studies have shown that structural and lexical features play an important role in recommending exception handling code examples (Rahman and Roy, 2014). In addition, Yu et al. (2012) suggested verbs of method names through machine learning based approaches and achieved promising results. Inspired by these studies, we employ deep learning technology to analyze the structure of method names for structural analysis and leverage the programming context information for lexical analysis to generate and recommend method names. To rename inconsistent method names, Liu K et al. (2019) extracted the feature representation

Fig. 2 User queries and commits related to flawed method names

of method names and method bodies through convolutional neural networks (CNNs) and suggested new method names with a recommendation list by searching similar names in a large method corpus. In addition, Allamanis et al. (2016) introduced a neural network that could predict a short and descriptive name for a code snippet. However, these approaches do not consider the structure of method names or fully leverage the programming context. Hence, there is much room for improvement in these approaches.

In this study, we propose a novel method renaming approach through structural and lexical analyses. This approach relies on a large method corpus and contains several components, including data pre-processing, structural analysis, lexical analysis, and renaming generation and recommendation. Our approach first leverages Word2Vec (https://code.google.com/p/word2vec/) (Mikolov et al., 2013) and CNN (Kim Y, 2014) to extract deep vector representations of method names and corresponding bodies, respectively. Then, for a given method name that has been identified as nonstandard or inconsistent with its method body, we find the possible verb-tokens and noun-tokens to generate a series of new method names. Specifically, we calculate the semantic similarity between the current method and methods in a large method corpus and find the possible correct verb-tokens from the most similar method names. At the same time, we search the current method body to identify possible correct noun-tokens. Finally, we combine the retrieved noun-tokens and verb-tokens to generate a series of new method names and suggest these method names using a designed recommendation algorithm.

To evaluate our proposed approach, we have performed extensive experiments with 90 824 methods of training data and 2111 methods with changed names of test data, which were collected from 20 open-source Java projects. Experimental results showed that the approach can achieve better results than the approaches proposed by Allamanis et al. (2016) and Liu K et al. (2019). For example, our approach achieved a Hit@5 of 33.62%, while the baseline approaches proposed by Allamanis et al. (2016) and Liu K et al. (2019) achieved only a Hit@5 of 1.44% and 19.50%, respectively. We also evaluated the impact of weight-adjusting noun-tokens and verb-tokens on generating and recommending new method names. Experimental results showed that our approach is insensitive to the value of the weight. We also validated the effectiveness of combining the structural and lexical analyses as well as our renaming generation and recommendation algorithm. Experimental results showed that our approach achieved better results than both of its variants. For example, our approach achieved a Hit@1 of 23.67% as a whole, but a Hit@1 of 2.18% and 8.13% with only the structural analysis and only the lexical analysis, respectively. As for the performance of the renaming generation and recommendation algorithm, our approach achieved the best results compared with using one naming style, i.e., camel case and underscore. For example, our approach achieved a Hit@1 of 23.67% as a whole, but a Hit@1 of 21.00% and 8.13% with the camel case and underscore styles, respectively. Finally, we conducted an empirical study to investigate the practicability of our approach in a real-world scenario by manually annotating the suggested method names. Experimental results showed that the suggested method names of our approach can ease the workload of developers in practice, especially for those developers who are new, inexperienced, or unfamiliar with the source code. The main contributions of this study are as follows:

1. We propose a novel method renaming approach that leverages structural analysis and lexical analysis to suggest high-quality method names for flawed method names. The implemented code is publicly available (https://github.com/Luojpljp/Method-Renaming).

2. We fully leverage method name structure and lexical analysis with programming context information to discover verb-tokens and noun-tokens to generate high-quality method names.

3. We have conducted a series of experiments to validate the effectiveness of our approach. The results demonstrated that our approach can significantly improve the state-of-the-art approaches.

2 Motivation

In this section, we discuss the motivation for this study, including the structure of method names and programming context.

2.1 Structure of method names

A good method name should help developers easily understand the main function of the method through its literal meaning. According to the Java programming specifications, a good method name should be formed using verbs or verb phrases (usually verbs combined with nouns) (Yu et al., 2012). To verify whether developers follow such code conventions in practice, we conducted an empirical study. We collected 87 003 Java methods from GitHub with at least 100 commits to ensure that these methods were well maintained. For these methods, we analyzed the part of speech (PoS) of their constitutive terms. Table 1 shows the empirical results. We found that 74.77% of the method names we collected were verb phrases, 22.50% were verbs, 1.54% were only nouns, and 1.19% of method names (hereinafter referred to as Other in Table 1) were very complex, e.g., containsAtLeast_ElementsIn_inOrder_success. The 1.19% of method names called "Other" were not taken into consideration in our study, since they may have had more than one verb, which makes the name ambiguous to understand the functions of these methods. There were 98.81% of method names containing verbs or nouns according to Table 1. As a result, we split the method names into two parts, i.e., verb-tokens, which represent the method action, and noun-tokens, which represent the object of the method. Based on this observation, we can generate new method names by combining verb-tokens with noun-tokens.

Table 1 The empirical results of the structure of method names

Classification	Exact number	Percentage (%)
Verb phrases	65 049	74.77
Verbs	19 573	22.50
Nouns	1344	1.54
Other	1037	1.19

2.2 Programming context data

It is common to find flawed method names in source code as a result of developer negligence (Liu K et al., 2019). In addition, methods with flawed names may have important functions and complex call relationships. In fact, the correct constitutive terms of flawed method names are usually hidden in the programming context. For example, in some *get* methods, the nouns in method names are hidden in the return statements. Hence, the programming context is an important source of information for constructing good method names, since it contains many useful tokens that can be used directly. However, programming context is not effectively used by existing approaches and needs further deep analysis. For example, the baseline approaches employed in this study do not fully consider the programming context. In contrast, these approaches ignore the useful tokens that can be used directly, to obtain similar or even the same method names in a large method corpus. However, there may be no method with the same name in the method corpus in some situations. In addition, finding a similar method name in a large method corpus is difficult, since there may be a lot of noise. For methods, we consider their method bodies along with annotations (if any) as the programming context. We study the feature extraction techniques of the programming context to find possible noun-tokens for generating high-quality method names.

3 Approach

In this section, we present the details of our method renaming approach. As illustrated in Fig. 3, it consists of four phases, i.e., data pre-processing, structural analysis, lexical analysis, and renaming generation and recommendation. The data pre-processing phase takes the method whose name needs to be renamed as input. In this phase the method name and tokens are split in the method body, prepared for the following phases. In the structural analysis phase the possible correct verb-tokens are found by calculating semantic similarities between the given method and a large method corpus using Word2Vec and CNN. Meanwhile, in the lexical analysis phase the possible correct noun-tokens are found using Word2Vec from the method body. In this way, we can obtain a series of verb-tokens and noun-tokens. Finally, in the renaming generation and recommendation phase, we combine the obtained verb-tokens and noun-tokens following their original styles, to generate a list of new recommended method names.

3.1 Data pre-processing

Our approach relies on a large-scale method corpus to find the constitutive terms to generate new

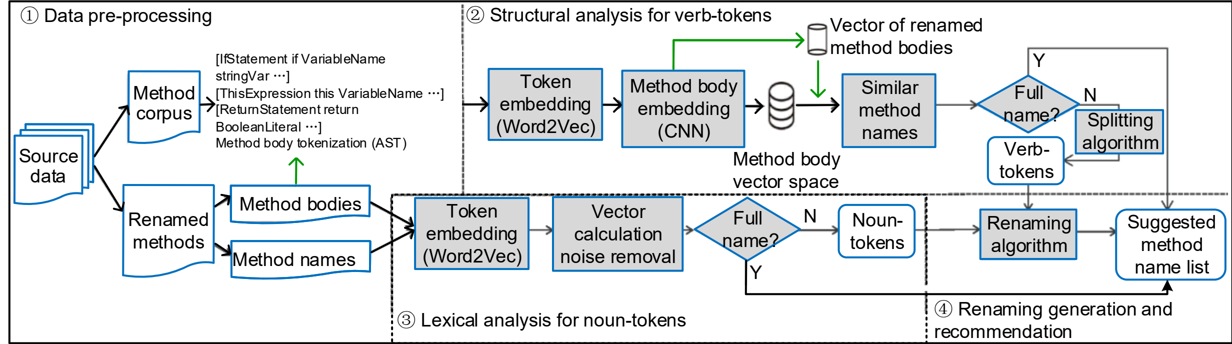

Fig. 3 The overall framework of our approach

names for those methods that need renaming. A given method that needs renaming is divided into two parts, i.e., a method name and a method body. Because the method body consists of multiple tokens that describe the method implementation, we convert it into a token sequence by following the paring method proposed by Liu K et al. (2019). This paring method converts the method body into an abstract syntax tree (AST) with a depth-first search algorithm. As a result, this paring method collects two kinds of tokens: AST node types and raw code tokens. For example, the declaration statement "String a;" containing two tokens will be converted into a four-token sequence [PrimitiveType, String, variable, a], which means that String is a primitive type and a is a variable in a statement. Because non-descriptive local variable names, such as a and b, can interfere with identifying similar code, all local variables are renamed as the concatenation of their data type with the string Var. As a result, the above declaration statement will be finally represented by the sequence [PrimitiveType, String, variable, StringVar].

For all the methods in the method corpus, we employ the same paring method to process their method bodies. In this way, the given method, which needs renaming, and all the methods in the method corpus, are processed using the same procedures.

3.2 Structural analysis

The structural analysis phase consists of three main components, embedding and normalizing method tokens, embedding method bodies, and obtaining verb-tokens. In the following, we present the details of these components.

3.2.1 Embedding and normalizing method tokens

For a given method that needs renaming and the large method corpus, we first embed all the tokens of these method bodies in vectors using Word2Vec, based on 100 billion words from Google News trained by Mikolov et al. (2013) and all tokens obtained from the training set of Liu K et al. (2019). Because a method body usually consists of a series of tokens, the method body is eventually represented as a two-dimensional numerical vector. We can convert a method body b into a sequence of tokens $T_b = (t_1, t_2, ..., t_k)$, where t_i is the token in location i, and further, into a two-dimensional numerical vector V_b as follows:

$$V_b \leftarrow l(T_b, \mathrm{TV_{NB}}), \tag{1}$$

where l is a function that integrates individual numerical vectors into a two-dimensional numerical vector based on the mapping function $\mathrm{TV_{NB}}$. Thus, we have the set of two-dimensional numerical vectors $V_\mathrm{B}, V_b = (v_1, v_2, ..., v_k) \in V_\mathrm{B}$, where $v_i \leftarrow \mathrm{TV_{NB}}(t_i)$.

$$\mathrm{TV_{NB}} \leftarrow F_\mathrm{W}(T_\mathrm{NB}), \tag{2}$$

where T_NB is a set of method names and method body token sequences as input for the token embedding function F_W (i.e., Word2Vec). The output is the token mapping function $\mathrm{TV_{NB}} : V_\mathrm{NB} \leftarrow \mathrm{TW_{NB}}$, where $\mathrm{TW_{NB}}$ is a term in method names and method bodies, and V_NB represents the numerical vector space embedding method names and tokens in $\mathrm{TW_{NB}}$.

Because the input layer of the CNN requires fixed vector sizes, we follow the workaround tested by Wang et al. (2016) to append zero vectors to make all vectors have the same size (the size of the longest

token sequence in our dataset). The left side of Fig. 4 shows a two-dimensional $n \times k$ numerical vector that represents a method body, where n is the vector size of each token and k is the size of the longest method body token sequence. Each row represents a vector of an embedded token, and the last two rows represent the appended zero vectors to make all two-dimensional vectors have the same size.

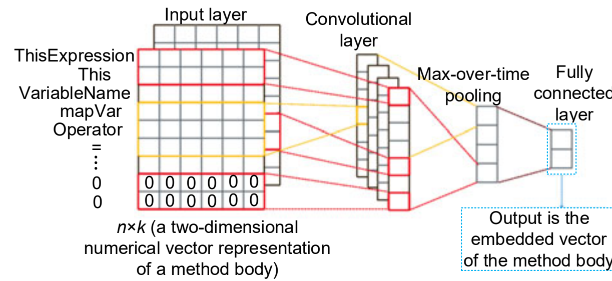

Fig. 4 The architecture of the convolutional neural network (CNN)

3.2.2 Embedding method bodies

The embedded token vectors are then fed into the CNN to embed all method bodies in numerical vectors. For method bodies, we design a map function to feed a two-dimensional numerical vector for each method body into the CNN, which is as follows:

$$\mathrm{VV}_{\mathrm{body}} \leftarrow F_{\mathrm{BV}}(V_{\mathrm{B}}), \tag{3}$$

where F_{BV} is an embedding function (i.e., NN) that takes the two-dimensional numerical vector bodies (V_{B}) as input and produces a map function ($\mathrm{VV}_{\mathrm{body}}$). $\mathrm{VV}_{\mathrm{body}}$ is defined as $\mathrm{VV}_{\mathrm{body}} : V_{\mathrm{B}}' \leftarrow V_{\mathrm{B}}$, where V_{B}' is an embedded vector space of method bodies. Based on $\mathrm{VV}_{\mathrm{body}}$, the body map function $\mathrm{NV}_{\mathrm{body}}$ is defined as

$$\mathrm{NV}_{\mathrm{body}} : V_{\mathrm{B}}' \leftarrow T_b, \tag{4}$$

where $\mathrm{NV}_{\mathrm{body}}$ is the composition of l and $\mathrm{VV}_{\mathrm{body}}$. $\mathrm{NV}_{\mathrm{body}}$ takes a token sequence of a method body and returns an embedded vector that represents it.

Fig. 4 shows the CNN architecture that our approach uses as embedding function F_{BV}. The input is two-dimensional numerical vectors of method bodies. The convolutional layer and max-over-time pooling layer are used to capture local features of methods and reduce the dimension of the input data. The network layers from the pooling layer to the fully connected layer are fully connected, and can combine all local features captured by the convolutional and pooling layers. We choose the output of the fully connected layer as vector representations of method bodies that synthesize all local features captured by the previous layers.

We can easily find the name corresponding to a given method's body vector in the name vector space, and vice versa. These linking relationships help us easily obtain the corresponding method name after finding a similar body for a given method.

3.2.3 Obtaining verb-tokens

For a given method that needs renaming, after embedding its method body in a vector, we can compute the cosine similarity between its vector and vectors of method bodies of the large method corpus (Liu K et al., 2019).

We first pick the top-10 methods from the corpus based on method body similarity. Then, we split those method names into several constitutive terms through a method name splitting algorithm. The splitting algorithm starts by scanning the first letter of each method name and works its way up to the verb that appears in the dictionary constructed from all the verbs of the corpus. Finally, we analyze the PoS tags of each constitutive term to extract the first verb-token (plus prepositions if the first verb token is followed by a preposition) and regard the first verb-token as the potentially correct verb-token for the given method that needs renaming.

3.3 Lexical analysis

The lexical analysis phase consists of two main components, i.e., embedding method names and obtaining noun-tokens. Specifically, for the given method that needs renaming, we first embed its method name in vectors using Word2Vec. Then, we compute the cosine similarity between the method name and all the body tokens with their embedding representation. Finally, we regard the noun-tokens with the largest similarity as the possible correct noun-tokens.

3.3.1 Embedding method names

As shown in Fig. 3, the given method name is first embedded in individual numerical vectors. The

embedding model is built as follows:

$$\mathrm{TV}'_{\mathrm{NB}} \leftarrow F_{\mathrm{W}}(T'_{\mathrm{NB}}), \tag{5}$$

where T'_{NB} is a set of method names as input to the token embedding function F_{W} (i.e., Word2Vec). The output is the token mapping function $\mathrm{TV}'_{\mathrm{NB}}$: $V'_{\mathrm{NB}} \leftarrow \mathrm{TW}'_{\mathrm{NB}}$, where $\mathrm{TW}'_{\mathrm{NB}}$ is a vocabulary of method names, and V'_{NB} represents the numerical vector space embedding method names in $\mathrm{TW}'_{\mathrm{NB}}$.

3.3.2 Obtaining noun-tokens

After embedding method names in vectors, we can compute the cosine similarity between method names and tokens to produce noun-tokens with the embedding representation. For a given method, we filter out several types of noisy tokens in its body, including programming keywords (e.g., return and String), API functions, punctuation marks, meaningless numbers, and single letters. In addition, we delete tokens that appear only once to avoid data explosion.

After filtering out noisy tokens from the method body, we compute the cosine similarity between the method name and all the method tokens. We select the top-10 noun-tokens that are most similar to the method name and regard them as the potentially correct noun-tokens for constructing the new method name.

3.4 Renaming generation and recommendation

This phase consists of two components, i.e., renaming generation and recommendation. First, we combine verb-tokens obtained from the structural analysis phase and noun-tokens obtained from the lexical analysis phase to generate a series of new method names. Second, we rank all the newly generated method names and form a recommendation list. The whole process is shown in Algorithm 1, which regards verb-tokens and noun-tokens and the method needing renaming as input, and outputs the new method name recommendation list.

3.4.1 Renaming generation

When we obtain noun-tokens N and verb-tokens V, we can calculate their similarity (lines 1 and 2) at the same time. Then, we identify the original formatting style of the method that needs renaming, because we follow the original style to format newly generated method names (Hindle et al., 2012). As shown in line 3 in Algorithm 1, given the method needing renaming M_{r}, we define the Style function, which obtains the formatting style of M_{r}. $\mathrm{Sty}_{\mathrm{m}}$ is the obtained original formatting style, i.e., camel case or underscore. Next, we normalize noun-tokens and verb-tokens into lower case (line 4). Finally, we can combine noun-tokens and verb-tokens through the generate function as follows (lines 5 to 9):

$$M_{\mathrm{list}} = \mathrm{generate}(N'_i, V'_j, \mathrm{Sty}_{\mathrm{m}}), \tag{6}$$

where generate() is the renaming generation function taking normalized noun-tokens N'_i, normalized verb-tokens V'_j, and $\mathrm{Sty}_{\mathrm{m}}$ as input. The generate function follows the original PoS sequence and the original style $\mathrm{Sty}_{\mathrm{m}}$ to combine N'_i and V'_j. In this way, we can generate a series of new method names, M_{list}.

3.4.2 Recommendation

After generating a series of new method names, we calculate a final score for each of the new method names as follows (line 10):

$$S_{\mathrm{m}} = \alpha S_{\mathrm{n}} + (1-\alpha) S_{\mathrm{v}}, \tag{7}$$

where S_{n} is the similarity score of a noun-token, S_{v} is the similarity score of a verb-token, and α is a weight to balance the contribution of the noun-token and verb-token. The output is the final score of each newly generated method name. Then we construct maps between these new methods and their corresponding final scores (line 11). Based on these maps, we can rank the newly generated method names and

Algorithm 1 Renaming generation and recommendation

Input: noun-tokens $N = \{N_1, N_2, ..., N_s\}$, verb-tokens $V = \{V_1, V_2, ..., V_t\}$, and the renaming method M_{r}
Output: recommendation list Top[k]
1: $S_{\mathrm{n}} = \mathrm{Sim}(N)$
2: $S_{\mathrm{v}} = \mathrm{Sim}(V)$
3: $\mathrm{Sty}_{\mathrm{m}} = \mathrm{Style}(M_{\mathrm{r}})$
4: $(N', V') = \mathrm{Normalize}(N, V)$
5: **for** $i = 0$ to s **do**
6: **for** $j = 0$ to t **do**
7: $M_{\mathrm{list}} = \mathrm{generate}(N'_i, V'_j, \mathrm{Sty}_{\mathrm{m}})$
8: **end for**
9: **end for**
10: $S_{\mathrm{m}} = \alpha S_{\mathrm{n}} + (1-\alpha) S_{\mathrm{v}}$
11: $\mathrm{Map}(M_{\mathrm{list}}, S_{\mathrm{m}})$
12: $\mathrm{Top}[k] = \mathrm{Rank}(\mathrm{Map}(M_{\mathrm{list}}, S_{\mathrm{m}}), k)$

select the top-k method names to recommend, where we vary k from 1 to 5 (line 12).

4 Experimental setup

In this section, we describe the research questions (RQs) we want to investigate, the baseline approaches for comparison, the evaluation metrics used in our study, the data collection procedures to obtain the large method corpus and the testing methods that need renaming, and the default parameter settings in our approach. The data collection and default parameter settings are presented in the appendix.

4.1 Research questions

RQ1: What is the impact of weight α adjusting noun-tokens and verb-tokens on generating and recommending new method names?

When generating and recommending new method names, we employ a weight α to adjust the importance of noun-tokens and verb-tokens as shown in Eq. (7). The weight balances the contribution of noun-tokens and verb-tokens, which could influence our approach in recommending new method names. To explore such influence, we set up this RQ.

To explore this RQ, we varied the weight α from 0.1 to 0.9 with a step of 0.1. For each specific value of α, we calculated the final value of each newly generated method name and further obtained the results of Hit@5 achieved by our approach to show its influence.

RQ2: Does combining the structural analysis and lexical analysis achieve better performance?

In our proposed approach, we design a structural analysis phase and a lexical analysis phase to obtain verb-tokens and noun-tokens, respectively. In this way, we can combine these verb-tokens and noun-tokens to generate a series of new method names for recommendation. The two phases are important to the performance of our approach. To evaluate how much structural analysis and lexical analysis contribute to our overall approach, we set up this RQ.

More precisely, we used the results of one of the two phases as the final recommended results for comparison. Specifically, we defined two variants of our approach. The first variant considered only the structural analysis results. In contrast, the second variant considered only the results of lexical analysis. We compared our whole approach with the two variants to investigate this RQ.

RQ3: Is the renaming generation and recommendation phase effective in our approach?

We design a renaming generation and recommendation phase in our proposed approach. This phase combines noun-tokens and verb-tokens to generate and rank a series of new method names. By this RQ, we aim to investigate whether this phase is effective.

According to the Java programming specification, it is suggested that method names follow the camel case style (Gosling et al., 2005; Li et al., 2021). However, method names and styles are usually project-specific (Hindle et al., 2012). Hence, we could not generalize all the method names using a specific naming style. We need to name methods dynamically. Our approach employs a dynamic way to integrate noun-tokens and verb-tokens to generate new method names following their original styles. To show the effectiveness of this phase, we used specific style, i.e., camel case or underscore, to form the new method names. By comparing this dynamic phase with a single style, we showed the effectiveness of the dynamic phase.

RQ4: To what extent is our approach superior to the baseline approaches in suggesting method names?

The method renaming approaches proposed by Allamanis et al. (2016) and Liu K et al. (2019) are the most recent for method renaming and achieve excellent results, so we have employed these approaches as the baselines. By this RQ, we want to investigate how much improvement our approach can achieve compared with the baseline approaches.

Liu K et al. (2019) opened the token sequences of methods in their training set to the public only. However, they did not detail which projects they used in their approach. Hence, we collected target projects from four high-quality communities: Apache, Google, Spring, and Hibernate. In addition, we implemented the approach of Allamanis et al. (2016) and validated it on our dataset. Hence, we employed our dataset to validate our approach and the baseline approaches for a fair comparison.

RQ5: To what extent can the suggested method names generated by our approach benefit developers?

Our proposed approach is expected to help developers rename flawed method names in real-world scenarios. To explore the extent to which the method names suggested by our approach can benefit developers, we set up this RQ.

Specifically, we conducted an empirical study to investigate the practicability of our approach in a real-world scenario. For this experiment, we first randomly sampled 10% (i.e., 211 methods) of the test methods. For these sampled test methods, we collected the top-1 suggested method names using our approach and their target method names. Then we recruited three volunteers to manually evaluate and annotate these sampled methods. For the evaluation, these volunteers were required to grade two scores for each sampled method, i.e., portability, which means to what extent the top-1 suggested method name could be adapted to the target method name, and consistency, which means whether the top-1 suggested method name was consistent with the corresponding method body. Both the portability and consistency grading scores ranged from 0 to 10, with a higher grading score representing a higher portability or consistency. For each sampled method, three volunteers independently graded its portability and consistency, and their average value was regarded as the final value. By collecting the annotation results from these volunteers, we determined the effectiveness of our approach in the real-world scenario.

4.2 Baseline approaches

The approach proposed by Allamanis et al. (2016) works as follows. A CNN was introduced with an attention mechanism. This approach uses a set of convolutional layers (without any pooling) to detect patterns in the input and identify "interesting" locations where attention should be focused. The approach can then predict a short and descriptive name for a code snippet (e.g., a method body) based on a trained network. The approach includes two sub-models: conv_attention, which uses only the pre-trained vocabulary, and copy_attention, which can copy tokens of input vectors (i.e., tokens in a method body). Experimental results showed that this approach can achieve an average Hit@1 of 0.57% and Hit@5 of 1.44%.

The approach of Liu K et al. (2019) works as follows. For a given method and method corpus, this approach first embeds method bodies in vectors. Then, it searches similar method bodies for the given method in the vector space of the method corpus. Next, it identifies names of these similar method bodies and introduces four ranking strategies to obtain four lists of similar method names. Strategy 1 relies solely on the similarities between method bodies. Strategy 2 first groups the same names and then ranks distinct names based on the sizes of the associated groups. Strategy 3 ranks the groups based on the average similarity, but the group sizes are not considered. Strategy 4 eventually re-ranks all groups produced in strategy 3 by downgrading all groups with only one instance to the lowest position. In this way, this approach can recommend new names for methods that need renaming. Experimental results showed that this approach can achieve a Hit@1 of 11.5% and a Hit@5 of 19.5%.

4.3 Evaluation metrics

We employed the same evaluation metrics as in related studies, i.e., the Hit Ratio, to evaluate the performance of different method renaming approaches (Yu et al., 2012; Liu K et al., 2019). Specifically, the Hit Ratio is calculated as follows:

$$\text{Hit Ratio} = \frac{\text{No. of correctly recommended names}}{\text{No. of all method names}}. \tag{8}$$

In this study, the recommended size k ranges from 1 to 5. We use Hit@k to demonstrate the Hit Ratio when recommending k results. For example, Hit@1 means the Hit Ratio when recommending one new method name.

5 Experimental results

In this section, we illustrate the detailed experimental results for the established RQs to show the effectiveness of our proposed approach.

5.1 Investigation of RQ1

RQ1: What is the impact of weight α adjusting noun-tokens and verb-tokens on generating and recommending new method names?

Fig. 5 shows the boxplot of the Hit@5 achieved by our approach in all projects, with the weight α varying from 0.1 to 0.9. From this figure, we can see that the weight α did not significantly impact the performance of our approach on the whole. This

means that our approach was insensitive to the value of the weight α. When α was equal to 0.5, the median Hit@5 in all projects was slightly better than the other values of α. For example, when α equaled 0.5, the median Hit@5 for all projects was 34.73%, which is better than those at other α values, varying from 33.81% to 34.56%. In other words, giving noun-tokens and verb-tokens the same weight (i.e., 0.5) is reasonable. Hence, in the following RQs, we set the weight α to 0.5 by default.

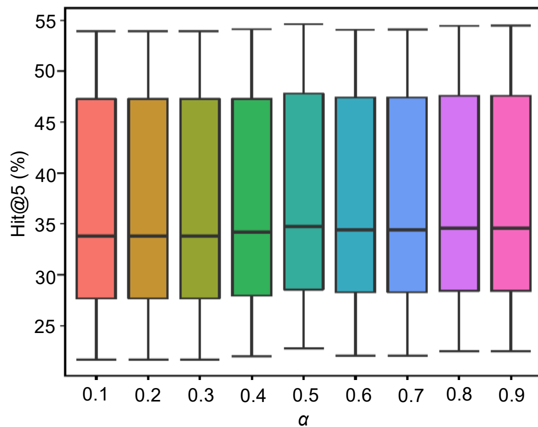

Fig. 5 The results of our approach when adjusting the weight α

Conclusion of RQ1 Our approach is insensitive to the weight α. When it is set to 0.5, our approach achieves the best results.

5.2 Investigation of RQ2

RQ2: Does combining the structural analysis and lexical analysis achieve better performance?

Fig. 6 shows the comparison results of our approach against its two variants. We can see that there were upward trends with the increase of k for our approach and its two variants. For example, Str achieved a Hit@1 of 2.18%. When k increased to 5, Str achieved a Hit@5 of 5.22%.

Our approach achieved obviously better results than both of its variants. For example, our approach achieved a Hit@1 of 23.67%, while Str and Lex achieved only a Hit@1 of 2.18% and 8.13%, respectively. When k was equal to 5, the disparity between our approach and its variants was even larger. For instance, our approach achieved a Hit@5 of 33.62%. In contrast, Str and Lex achieved only a Hit@5 of 5.22% and 10.44%, respectively.

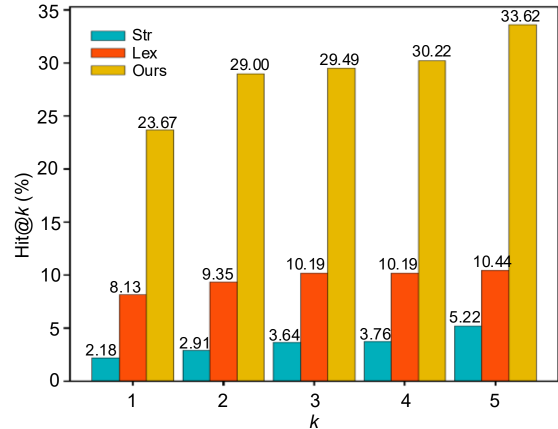

Fig. 6 The comparison results of our approach and its two variants

Str stands for the variant that employs only structural analysis in our approach. Lex is the variant that employs only lexical analysis. Ours shows our approach using both structural analysis and lexical analysis

The reason why our approach can achieve better results than its two variants may be that we conducted an empirical study to explore the composition of method names from the perspective of PoS. The empirical results show that most of the method names consist of both noun-tokens and verb-tokens. Based on these empirical results, we have designed a new method name generation and recommendation algorithm. Hence, our approach is superior to its two variants.

Conclusion of RQ2 When combining the two phases, our approach achieves better results than its two variants.

5.3 Investigation of RQ3

RQ3: Is the renaming generation and recommendation phase effective in our approach?

We have designed a dynamic method generation and recommendation phase in our approach that can decide the final styles for the new method names based on their original styles. To validate the effectiveness of this phase, we compared it with two single styles, i.e., camel case and underscore. Fig. 7 shows the comparison results of our approach with the default phase and the two single styles. We can see similar and obvious upward trends with the increase of k for our approach with different method name formatting styles.

Ours achieved the best results compared to Under and Camel. For example, using the underscore style as the formatting style for new method

names, Under achieved only a Hit@1 of 8.13%, whereas Camel achieved a Hit@1 of 21.00%. In contrast, Ours, leveraging the dynamic formatting style, achieved a Hit@1 of 23.67% under the same conditions. This means that although the constitutive terms of method names were changed during the evolution of software projects, the formatting styles of method names remained unchanged.

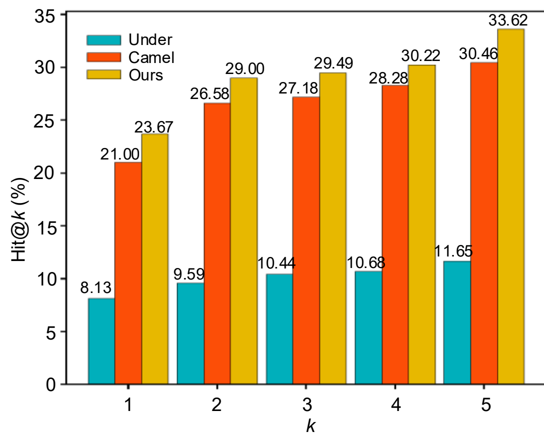

Fig. 7 The comparison results of our approach with the default phase and the two single styles

Under stands for our approach with the underscore style for formatting new method names. Camel is our approach with the camel case as the method name formatting style. Ours shows our approach with the default dynamic phase

Conclusion of RQ3 The renaming generation and recommendation phase that leverages the dynamic formatting style is effective in our approach.

5.4 Investigation of RQ4

RQ4: To what extent is our approach superior to the baseline approaches in suggesting method names?

Fig. 8 shows the comparison results of our approach against the baseline approaches proposed by Allamanis et al. (2016) and Liu K et al. (2019). Allamanis et al. (2016) introduced two sub-models (hereinafter referred to as conv and copy in Fig. 8) to suggest method names for a given method. From Fig. 8, we can see that our approach achieved much better results than both of the two sub-models introduced by Allamanis et al. (2016). For example, the best score of two sub-models was only 0.57% in terms of the Hit@1. In contrast, our approach can achieve a Hit@1 of 23.67%. When k increased to 5, our approach achieved a Hit@5 of 33.62%, whereas the best result of the two sub-models was 1.44%. Our approach performed much better than the baseline approach proposed by Allamanis et al. (2016) by 23.10% in terms of the Hit@1 and 32.18% in terms of the Hit@5. In summary, our approach achieved significantly better results than the approach proposed by Allamanis et al. (2016).

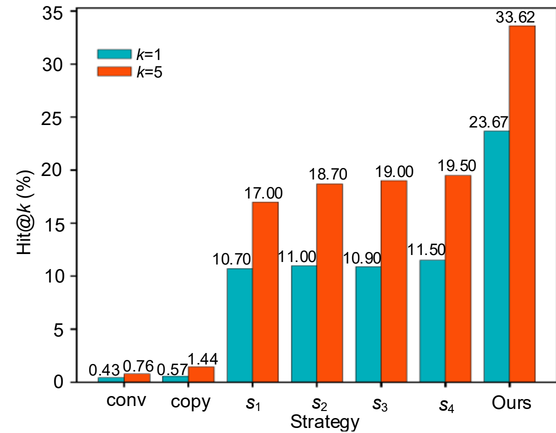

Fig. 8 The comparison results of our approach and baselines

Liu K et al. (2019) proposed four strategies (referred to as s_1, s_2, s_3, and s_4 in Fig. 8) to generate four recommendation lists of new method names. From Fig. 8, we can see that our approach achieved better results than all these four strategies. For example, our approach achieved a Hit@1 of 23.67%, whereas the strategy that performed the best among the four strategies in Liu K et al. (2019) achieved only a Hit@1 of 11.50%. When k increased to 5, our approach achieved a Hit@5 of 33.62%. In contrast, the best result achieved by the four strategies in Liu K et al. (2019) was only 19.50% under the same conditions. This means that our approach outperformed the baseline approach proposed by Liu K et al. (2019) by 14.12% in terms of Hit@5.

Recommending the same target method names is a very difficult research task, because the proposed approach should not only correctly generate all the constitutive terms for the correct method names, but also combine them in the correct sequence. In addition, the proposed approach should employ the correct formatting style when combining constitutive terms. Hence, even though our approach is superior to the state-of-the-art baseline approaches in terms of the Hit Ratio, our approach can achieve only an absolute Hit Ratio value of up to 33.62% when recommending five results. Recommending exactly the

same method name may be too strict. Hence, we changed the definition of the Hit Ratio and divided it into two types. The first type of Hit Ratio follows its original definition as shown in Eq. (8), We call this the Full Hit Ratio. In the second type of Hit Ratio, if the stemmed constitutive terms of the recommended new method name and the target method name are the same, we regard it as a Hit. We call this type a Part Hit Ratio. The Part Hit Ratio can also help developers generate exactly correct method names to a large extent.

Fig. 9 shows the results of the Full and Part Hit Ratios of our approach. The Part Hit Ratio achieved by our approach was better than the Full Hit Ratio, because the Part Hit Ratio is less strict than the Full Hit Ratio. For example, our approach achieved a Part Hit@1 of 26.94%, and achieved a Full Hit@1 of 23.67%. When k increased to 5, our approach achieved a Part Hit@5 of 37.14%. In contrast, our approach achieved a Full Hit@5 of 33.62%.

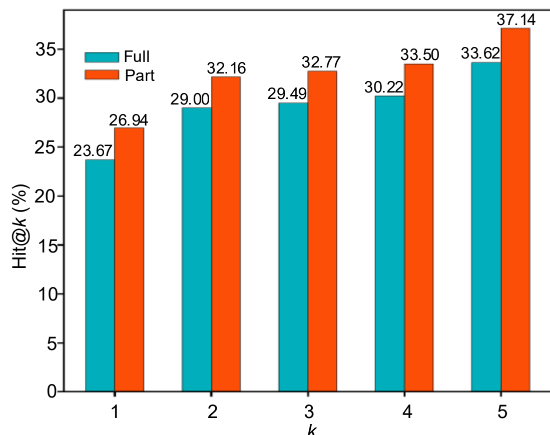

Fig. 9 The results of Full and Part Hit Ratios achieved by our approach

Full stands for the Full Hit Ratio and Part stands for the Part Hit Ratio

Conclusion of RQ4 Our approach is superior to the state-of-the-art baseline approaches in terms of the Hit Ratio.

5.5 Investigation of RQ5

RQ5: To what extent can the suggested method names generated by our approach benefit developers?

Table 2 shows the distribution of the grading scores for portability and consistency. As mentioned above, the grading score ranges from 0 to 10, with a larger grading score standing for a higher portability or consistency. We can see that more than a half of the scores were 10 for portability and consistency. For example, the score of 10 accounted for the largest percentage at 51.18% for portability and 53.08% for consistency. The scores of 4 and 0 accounted for the lowest percentage at 0.95% for portability and 0.00% for consistency. The sum percentage of those scores of 5 and greater was 88.63% for portability and 93.85% for consistency, which means that most of the method names suggested by our approach can be easily adapted to the correct target method names by developers and that these suggested method names were consistent with their method bodies. Hence, we think that the method names suggested by our approach can ease the workload of developers in practice, especially for developers who are new, inexperienced, or unfamiliar with the source code.

Table 2 The annotation results of sampled methods in terms of portability and consistency

Score	Portability		Consistency	
	Number	Percentage	Number	Percentage
0	9	4.26%	0	0.00%
1	3	1.42%	4	1.89%
2	3	1.42%	3	1.42%
3	7	3.32%	4	1.89%
4	2	0.95%	2	0.95%
5	25	11.85%	7	3.32%
6	23	10.90%	22	10.43%
7	9	4.27%	20	9.48%
8	17	8.06%	14	6.64%
9	5	2.37%	23	10.90%
10	108	51.18%	112	53.08%

Conclusion of RQ5 Most of the portability and consistency scores of suggested method names are relatively high, which means that the suggested method names can benefit developers.

6 Threats to validity

In this section, we discuss the threats to validity, both internal and external.

6.1 Threats to internal validity

One threat to internal validity is the limitation of parsing method names, because the composition of method names is very complex and some of them do not follow the naming convention. It is challenging to propose a uniform model to parse all kinds

of method names. This threat could be reduced by developing more advanced method name parsing tools with natural language processing. Another threat to internal validity is the test data, because the test data method names must be actually fixed to evaluate the performance of our approach. This threat could be reduced by employing more projects to guarantee a fixed number of method names.

6.2 Threats to external validity

A threat to external validity is the constructed corpus, because it is impossible to ensure that all methods in the corpus have standard and consistent names. To address this threat, we selected well-maintained open projects with a high reputation. In the future, we will employ more corpora to validate our approach. Another threat to external validity is the data explosion of method body tokens, because the longer the method bodies are, the less effective function they represent. To address this threat, we keep methods with at most 120 tokens. In the future, we will consider more methods with more tokens to validate our approach.

7 Related works

Naming or renaming in code has received a fair amount of research attention. There have been several studies focusing on code renaming (Butler et al., 2010, 2011, 2013; Butler, 2012). In this section, we roughly divide the related works into two categories, i.e., rule-based renaming and learning-based renaming.

7.1 Rule-based renaming

Rule-based renaming approaches recognize renaming opportunities and suggest effective names by enforcing formal naming convention rules (Allamanis et al., 2014; Liu H et al., 2015; Li et al., 2021).

Caprile and Tonella (1999, 2000) proposed a dictionary-based approach to standardize the lexicon of constitutive terms and the syntactic structure of function names. The approach identifies words that are not found in the standard dictionary but listed in the synonym dictionary as renaming opportunities. Abebe and Tonella (2013) selected candidate concepts and identified relationships through extracted ontology from the source code to generate suggested names. Corbo et al. (2007), Binkley et al. (2011), Butler (2016), and Kim S and Kim D (2016) identified identifier structures by checking PoS rules against naming conventions. Binkley et al. (2011) constrained PoS rules on field names. They extracted four PoS rules for field names and identified names that violate such rules as renaming opportunities.

Our work was motivated by these prior studies, which use formal naming conventions and dictionaries to identify renaming opportunities. The difference is that our work focuses on programming context data similar to the dictionary to obtain noun-tokens in preparation for generating high-quality method names.

7.2 Learning-based renaming

Learning-based approaches apply machine learning techniques, e.g., the n-gram statistical language model and CNN, to identify inconsistent method names and suggest effective names.

The n-gram statistical language model has been widely exploited to identify renaming opportunities (Hindle et al., 2012; Nguyen et al., 2013; Allamanis et al., 2014; Suzuki et al., 2014; Lin et al., 2017). NATURALIZE (Allamanis et al., 2014) pioneered the application of the n-gram statistical language model to check irregularities. It identifies unnatural identifiers based on the probability distribution of all textual tokens linearly scanned in the code document using a moving window. Allamanis et al. (2015, 2016) exploited a log-bi-linear neural network to suggest method and class names. Liu K et al. (2019) exploited the paragraph vector (Le and Mikolov, 2014) and CNN (Matsugu et al., 2003) to identify inconsistent method names and suggest better ones. The evaluation results suggested that the F-measure of the approach in identifying inconsistent method names was 68%.

Our work was inspired by these prior studies that use machine learning techniques, e.g., Word2Vec and CNN. However, our work differs from these studies in that our work considers the structure of method names to obtain verb-tokens in preparation for generating high-quality method names.

8 Conclusions and future work

Methods are the basic units of source code and method names are the keys to the understandability and maintenance of methods. However, giving methods appropriate names is one of the most difficult tasks for developers. Hence, it is common to find a lot of flawed (e.g., non-standard or inconsistent) method names, which may lead to software defects. To help developers eliminate flawed names, we propose a novel approach to suggest effective names for methods using structural analysis and lexical analysis. Experimental results showed that our approach significantly outperforms the state-of-the-art baseline approaches in terms of the Hit Ratio.

For future work, we have the following directions. First, we plan to validate our approach in more corpora to generalize its performance. Second, we plan to extend our approach to recommend new names for all the identifier categories, including methods, types, and fields. Third, we plan to build an automatic tool that encapsulates our approach to help developers rename methods.

Contributors

Junpeng LUO and Jingxuan ZHANG designed the research. Junpeng LUO processed the data. Junpeng LUO and Jingxuan ZHANG drafted the paper. Yong XU and Chenxing SUN helped organize the paper. Jingxuan ZHANG and Zhiqiu HUANG revised and finalized the paper.

Compliance with ethics guidelines

Junpeng LUO, Jingxuan ZHANG, Zhiqiu HUANG, Yong XU, and Chenxing SUN declare that they have no conflict of interest.

References

Abebe SL, Tonella P, 2013. Automated identifier completion and replacement. Proc 17th European Conf on Software Maintenance and Reengineering, p.263-272. https://doi.org/10.1109/CSMR.2013.35

Abebe SL, Haiduc S, Tonella P, et al., 2011. The effect of lexicon bad smells on concept location in source code. Proc 11th Int Working Conf on Source Code Analysis and Manipulation, p.125-134. https://doi.org/10.1109/SCAM.2011.18

Abebe SL, Arnaoudova V, Tonella P, et al., 2012. Can lexicon bad smells improve fault prediction? Proc 19th Working Conf on Reverse Engineering, p.235-244. https://doi.org/10.1109/WCRE.2012.33

Allamanis M, Barr ET, Bird C, et al., 2014. Learning natural coding conventions. Proc 22nd ACM SIGSOFT Int Symp on Foundations of Software Engineering, p.281-293. https://doi.org/10.1145/2635868.2635883

Allamanis M, Barr ET, Bird C, et al., 2015. Suggesting accurate method and class names. Proc 10th Joint Meeting on Foundations of Software Engineering, p.38-49. https://doi.org/10.1145/2786805.2786849

Allamanis M, Peng H, Sutton C, 2016. A convolutional attention network for extreme summarization of source code. Proc 33rd Int Conf on Machine Learning, p.2091-2100.

Amann S, Nguyen HA, Nadi S, et al., 2019. A systematic evaluation of static API-misuse detectors. *IEEE Trans Softw Eng*, 45(12):1170-1188. https://doi.org/10.1109/TSE.2018.2827384

Arnaoudova V, Eshkevari LM, di Penta M, et al., 2014. REPENT: analyzing the nature of identifier renamings. *IEEE Trans Softw Eng*, 40(5):502-532. https://doi.org/10.1109/TSE.2014.2312942

Binkley D, Hearn M, Lawrie D, 2011. Improving identifier informativeness using part of speech information. Proc 8th Working Conf on Mining Software Repositories, p.203-206. https://doi.org/10.1145/1985441.1985471

Butler S, 2012. Mining Java class identifier naming conventions. Proc 34th Int Conf on Software Engineering, p.1641-1643. https://doi.org/10.1109/ICSE.2012.6227216

Butler S, 2016. Analysing Java Identifier Names. PhD Thesis, the Open University, England Birmingham, UK.

Butler S, Wermelinger M, Yu YJ, et al., 2009. Relating identifier naming flaws and code quality: an empirical study. Proc 16th Working Conf on Reverse Engineering, p.31-35. https://doi.org/10.1109/WCRE.2009.50

Butler S, Wermelinger M, Yu YJ, et al., 2010. Exploring the influence of identifier names on code quality: an empirical study. Proc 14th European Conf on Software Maintenance and Reengineering, p.156-165. https://doi.org/10.1109/CSMR.2010.27

Butler S, Wermelinger M, Yu YJ, et al., 2011. Mining Java class naming conventions. Proc 27th IEEE Int Conf on Software Maintenance, p.93-102. https://doi.org/10.1109/ICSM.2011.6080776

Butler S, Wermelinger M, Yu YJ, et al., 2013. INVocD: identifier name vocabulary dataset. Proc 10th Working Conf on Mining Software Repositories, p.405-408. https://doi.org/10.1109/MSR.2013.6624056

Caprile B, Tonella P, 1999. Nomen est omen: analyzing the language of function identifiers. Proc 6th Working Conf on Reverse Engineering, p.112-122. https://doi.org/10.1109/WCRE.1999.806952

Caprile B, Tonella P, 2000. Restructuring program identifier names. Proc Int Conf on Software Maintenance, p.97-107. https://doi.org/10.1109/ICSM.2000.883022

Corbo F, del Grosso C, di Penta M, 2007. Smart formatter: learning coding style from existing source code. Proc IEEE Int Conf on Software Maintenance, p.525-526. https://doi.org/10.1109/ICSM.2007.4362682

Gosling J, Joy B, Steele G, et al., 2005. The Java™ Language Specification (3rd Ed.). Addison-Wesley, New York, USA.

Hindle A, Barr ET, Su ZD, et al., 2012. On the naturalness of software. Proc 34th Int Conf on Software Engineering, p.837-847. https://doi.org/10.1109/ICSE.2012.6227135

Høst EW, Østvold BM, 2009. Debugging method names. Proc 23rd European Conf on Object-Oriented Programming, p.294-317.
https://doi.org/10.1007/978-3-642-03013-0_14
Kim S, Kim D, 2016. Automatic identifier inconsistency detection using code dictionary. *Empir Softw Eng*, 21(2):565-604.
https://doi.org/10.1007/s10664-015-9369-5
Kim Y, 2014. Convolutional neural networks for sentence classification. Proc Conf Empirical Methods in Natural Language Processing, p.1746-1751.
https://doi.org/10.3115/v1/D14-1181
Lawrie D, Morrell C, Feild H, et al., 2006. What's in a name? A study of identifiers. Proc 14th IEEE Int Conf on Program Comprehension, p.3-12.
https://doi.org/10.1109/ICPC.2006.51
Le Q, Mikolov T, 2014. Distributed representations of sentences and documents. Proc 31st Int Conf on Machine Learning, p.II-1188-II-1196.
Li GJ, Liu H, Nyamawe AS, 2021. A survey on renamings of software entities. *ACM Comput Surv*, 53(2):41.
https://doi.org/10.1145/3379443
Lin B, Scalabrino S, Mocci A, et al., 2017. Investigating the use of code analysis and NLP to promote a consistent usage of identifiers. Proc 17th Int Working Conf on Source Code Analysis and Manipulation, p.81-90.
https://doi.org/10.1109/SCAM.2017.17
Liu H, Liu QR, Liu Y, et al., 2015. Identifying renaming opportunities by expanding conducted rename refactorings. *IEEE Trans Softw Eng*, 41(9):887-900.
https://doi.org/10.1109/TSE.2015.2427831
Liu K, Kim D, Bissyandé TF, et al., 2019. Learning to spot and refactor inconsistent method names. Proc 41st Int Conf on Software Engineering, p.1-12.
https://doi.org/10.1109/ICSE.2019.00019
Liu K, Kim D, Bissyandé TF, et al., 2021. Mining fix patterns for FindBugs violations. *IEEE Trans Softw Eng*, 47(1):165-188.
https://doi.org/10.1109/TSE.2018.2884955
Matsugu M, Mori K, Mitari Y, et al., 2003. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neur Netw*, 16(5-6):555-559.
https://doi.org/10.1016/S0893-6080(03)00115-1
Mikolov T, Chen K, Corrado G, et al., 2013. Efficient estimation of word representations in vector space.
https://arxiv.org/abs/1301.3781
Nguyen TT, Nguyen AT, Nguyen HA, et al., 2013. A statistical semantic language model for source code. Proc 9th Joint Meeting on Foundations of Software Engineering, p.532-542.
https://doi.org/10.1145/2491411.2491458
Rahman MM, Roy CK, 2014. On the use of context in recommending exception handling code examples. Proc 14th Int Working Conf on Source Code Analysis and Manipulation, p.285-294.
https://doi.org/10.1109/SCAM.2014.15
Suzuki T, Sakamoto K, Ishikawa F, et al., 2014. An approach for evaluating and suggesting method names using n-gram models. Proc 22nd Int Conf on Program Comprehension, p.271-274.
https://doi.org/10.1145/2597008.2597797
Takang AA, Grubb PA, Macredie RD, 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J Program Lang*, 4:143-167.
Wang S, Liu TY, Tan L, 2016. Automatically learning semantic features for defect prediction. Proc 38th Int Conf on Software Engineering, p.297-308.
https://doi.org/10.1145/2884781.2884804
White M, Tufano M, Vendome C, et al., 2016. Deep learning code fragments for code clone detection. Proc 31st IEEE/ACM Int Conf on Automated Software Engineering, p.87-98.
Yu SS, Zhang RC, Guan JH, 2012. Properly and automatically naming Java methods: a machine learning based approach. Proc 8th Int Conf on Advanced Data Mining and Applications, p.235-246.
https://doi.org/10.1007/978-3-642-35527-1_20

Appendix: Experimental setup

The data collection and default parameter settings are detailed in this appendix.

1. Data collection

In this subsection, we describe the detailed procedures for collecting the experimental data used in this study. In the literature, no one has provided an open benchmark dataset for the method renaming research task. Consequently, we constructed our own dataset and have opened it to the public (https://github.com/Luojpljp/Method-Renaming), with the hope of helping other researchers handle the same research task.

Specifically, we constructed this dataset following a series of procedures. First, we considered and selected a collection of open-source projects and regarded them as the target projects. We collected target projects from four high-quality communities, Apache, Google, Spring, and Hibernate, because they are relatively mature and many users and developers are using projects in these communities. For each community, we downloaded the top-10 Java projects.

Then, for each collected Java project, we parsed the commit history of each project to check whether there was a method name that had been changed using the git command, i.e., "git log -L," whose output is the trace result. If the size of the trace result was greater than one, the name of a specific method had been changed in at least one commit in history, and we kept it in the final testing dataset. Finally, 2111 methods remained in the benchmark testing dataset. For each method in the testing dataset, we needed

to return it to its historical version and regarded the historical version as the testing method. In this way, we could know the exact target new method names and better evaluate different approaches. We returned the testing methods to their historical version using the git command "git reset -hard." The historical version of these 2111 testing methods was used as the testing benchmark dataset.

Based on the collected benchmark dataset, we explored the potential reasons for method renaming. We first sampled one tenth of the methods from this dataset, i.e., 211 methods. By investigating their change history and the corresponding programming context, we manually generated the reasons for method renaming and further merged similar ones. Finally, as shown in Table A1, we summarized four reasons for method renaming, i.e., correcting function inconsistency (referred to as function inconsistency in Table A1), correcting object inconsistency (referred to as object inconsistency in Table A1), correcting programming specification inconsistency (referred to as PS inconsistency in Table A1), and correcting spelling mistakes (referred to as spelling mistakes in Table A1). Correcting function inconsistency means that the renaming was designed to modify the verb of the method name; e.g., the original method name was removeConfigurers, whereas the modified method name was changed to getConfigurers. Correcting object inconsistency means that the object of the method was changed in method renaming; e.g., the original method name was getUserBase, whereas the modified name was getRoleBase. Correcting programming specification inconsistency means making the method name consistent with the Java programming specification, and correcting spelling mistakes means that developers correct the accidentally written wrong word in method names.

We also calculated the percentage of each reason in the sample methods shown in Table A1. Correcting object inconsistency accounted for the largest proportion, i.e., 74.88%. The next largest proportion was correcting function inconsistency, which accounted for 18.01%. Correcting programming specification inconsistency accounted for 5.21%, and correcting spelling mistakes accounted for only 1.90%. This means that most method names were renamed because of correcting function inconsistency and correcting object inconsistency, which accounted for 92.89% in total.

Table A1 The reasons for method renaming

Classification	Exact number	Percentage (%)
Function inconsistency	38	18.01
Object inconsistency	158	74.88
PS inconsistency	11	5.21
Spelling mistakes	4	1.90

The methods not in the testing benchmark dataset in the selected projects were used to construct a large method corpus to facilitate new method name generation. For the remaining methods that were not in the testing dataset, we filtered out some of them based on method types. Specifically, we removed main methods, constructor methods, and empty methods without method body implementation, because they have less effect on program functionality and their method names seldom need to be changed. As a result, 104 107 methods were collected.

To avoid the reduction in the intelligibility of methods and the explosion of code tokens, we further limited the number of constitutive terms of method names (method name length) and the number of tokens in the method body (method body size). For each remaining method, we calculated the length of its name and the size of its body. Fig. A1 shows the boxplot of the size distribution of method bodies and the method name length for all the remaining methods. The method name length ranged from 1 to 9, while the method body size ranged from 2 to 1362. A total of 103 301 method names had fewer than 6 terms and 100 678 methods had fewer than 120 tokens. As a result, we kept only those methods whose name length was no more than 6 and whose method body size was no more than 120 tokens. Finally, 90 824 methods remained in the large method corpus.

2. Default parameters

As mentioned above, Word2Vec and CNN were used in our approach, and their parameters needed to be configured. In our approach, all the parameters for these two techniques were configured according to the parameter settings proposed by Kim Y (2014) and Liu K et al. (2021), because these approaches have been shown to achieve promising results. This means that the parameter settings were the same as

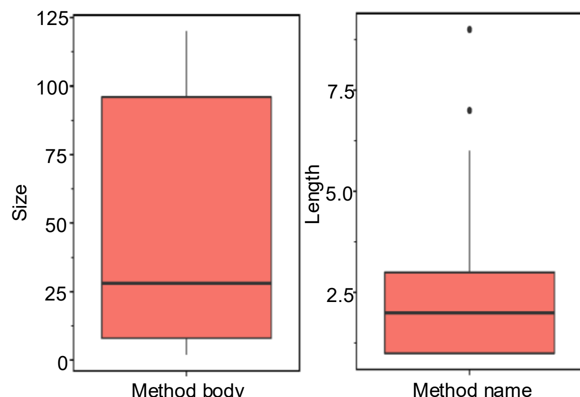

Fig. A1 The size distribution of method bodies and the length of method names

in Kim Y (2014) and Liu K et al. (2021). Tables A2 and A3 show the parameter settings in Word2Vec and CNN, respectively. For example, the size of the vector and the learning rate were set to 300 and 0.025, respectively, in Word2Vec. The activation of the output layer was set as softmax in CNN. In addition, both techniques were implemented with the open DL4J library (https://deeplearning4j.org/).

Table A2 Parameter settings of Word2Vec

Parameter	Value	Parameter	Value
Vector size	300	Window size	2
Mid word frequency	1	Learning rate	0.025

Table A3 Parameter settings of CNN

Parameter	Value
Learning rate	1×10^{-2}
Number of nodes in the hidden layer	1000
Pooling type	Max pool
Activation (other layers)	ReLU
Activation (output layer)	Softmax
Loss function	Mean absolute error
Optimization algorithm	Stochastic gradient descent