Learning to Spot and Refactor Inconsistent Method Names

Kui Liu[†], Dongsun Kim[†], Tegawendé F. Bissyandé[†], Taeyoung Kim[‡], Kisub Kim[†], Anil Koyuncu[†],
Suntae Kim[‡], Yves Le Traon[†]
[†]Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
{kui.liu, dongsun.kim, tegawende.bissyande, kisub.kim, koyuncu.anil, yves.letraon}@uni.lu
[‡]Department of Software Engineering, Chonbuk National University, South Korea
{rlaxodud1200, jipsin08}@gmail.com

***Abstract*—To ensure code readability and facilitate software maintenance, program methods must be named properly. In particular, method names must be consistent with the corresponding method implementations. Debugging method names remains an important topic in the literature, where various approaches analyze commonalities among method names in a large dataset to detect inconsistent method names and suggest better ones. We note that the state-of-the-art does not analyze the implemented code itself to assess consistency. We thus propose a novel automated approach to debugging method names based on the analysis of consistency between method names and method code. The approach leverages deep feature representation techniques adapted to the nature of each artifact. Experimental results on over 2.1 million Java methods show that we can achieve up to 15 percentage points improvement over the state-of-the-art, establishing a record performance of 67.9% F1-measure in identifying inconsistent method names. We further demonstrate that our approach yields up to 25% accuracy in suggesting full names, while the state-of-the-art lags far behind at 1.1% accuracy. Finally, we report on our success in fixing 66 inconsistent method names in a live study on projects in the wild.**

"If you have a good name for a method, you don't need to look at the body." — Fowler et al. [1]

I. Introduction

Names unlock the door to languages. In programming, names (i.e., identifiers) are pervasive in all program concepts, such as classes, methods, and variables. Descriptive names are the intuitive characteristic of objects being identified; thus, correct naming is essential for ensuring readability and maintainability of software programs. As highlighted by a number of industry experts, including McConnell [2], Beck [3], and Martin [4], naming is one of the key activities in programming.

Naming is a non-trivial task for program developers. Studies conducted by Johnson [5], [6] concluded that identifier naming is the hardest task that programmers must complete. Indeed, developers often write poor (i.e., inconsistent) names in programs due to various reasons, such as lacking a good thesaurus, conflicting styles during collaboration among several developers, and improper code cloning [7].

```
public boolean containsField(Field f){
    return fieldsList.contains(f);}

private ResolvedMember findField(ResolvedType resolvedType,String fieldName){
    for(ResolvedMember field : resolvedType.getDeclaredFields()){
        if (field.getName().equals(fieldName)){ return field;}}
    return null;}

public Field containsField(String name){
    for(Iterator e = this.field_vec.iterator(); e.hasNext();){
        Field f = (Field) e.next();
        if (f.getName().equals(name)){ return f;}}
    return null;}
```

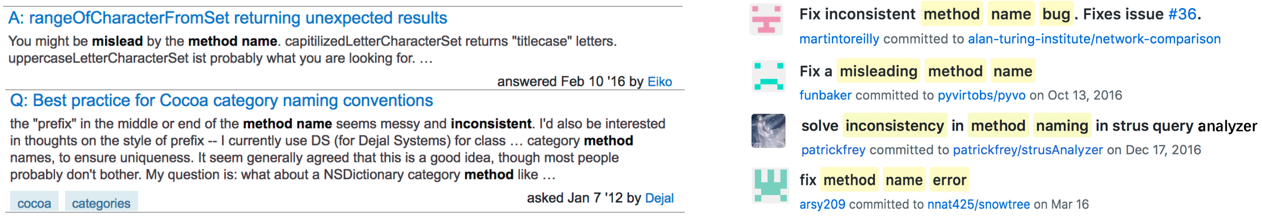
Fig. 1. Motivation examples taken from project `AspectJ`.

Method names are the intuitive and vital information for developers to understand the behavior of programs or APIs [8]–[11]. Therefore, inconsistent method names can make programs harder to understand and maintain [12]–[18], and may even lead to software defects [19]–[22]. Poor method names are indeed prone to be defective. For example, the commonly-used FindBugs [23] static analyzer even enumerates up to ten bug types related to method identifiers.

Figure 1 provides examples from project `AspectJ` [24] to illustrate how inconsistent names can be confusing about the executable behavior of a method. The name of the first method, `containsField`, suggests a question and is consistent with the method behavior which is about checking whether the `fieldsList` contains the target field `f`. The second method implements the search of a field in the target dataset and is thus consistently named `findField`. The third method is implemented similarly to the second method `findField`, but is named `containsField` as the first method. This name is inconsistent and can lead to misunderstanding of API usage.

As a preliminary study on the extent of the inconsistent method naming problem, we investigated posts by developers and users on fora and code repositories. We performed a search using composite conjunctions of "*method name*" and a category of keywords (i.e., *inconsistent, consistency, misleading, inappropriate, incorrect, confusing, wrong, bug* and *error*) to match relevant questions in StackOverflow [25] and commit logs in GitHub [26]. As a result, we managed to spot 5,644 questions and 183,901 commits. Figures 2 show some excerpts of retrieved results.

Additionally, to assess the extent to which developers are prone to fix method names, we investigated the history of changes in all 430 projects collected for our experiments: in 53,731 commits, a method name is changed without any change to the corresponding body code. We further tracked future changes and noted that in 16% of the cases, the change is final (i.e., neither the method body nor the method name is changed again in later revisions of the project). These findings suggest that developers are indeed striving to choose appropriate method names, often to address consistency with the contexts of their code.



A: rangeOfCharacterFromSet returning unexpected results

You might be **mislead** by the **method name**. capitilizedLetterCharacterSet returns "titlecase" letters. uppercaseLetterCharacterSet ist probably what you are looking for. …

answered Feb 10 '16 by Eiko

Q: Best practice for Cocoa category naming conventions

the "prefix" in the middle or end of the **method name** seems messy and **inconsistent**. I'd also be interested in thoughts on the style of prefix -- I currently use DS (for Dejal Systems) for class … category **method** names, to ensure uniqueness. It seem generally agreed that this is a good idea, though most people probably don't bother. My question is: what about a NSDictionary category **method** like …

cocoa categories

asked Jan 7 '12 by Dejal

(a) From questions in StackOverflow.

Fix inconsistent method name bug. Fixes issue #36.

martintoreilly committed to alan-turing-institute/network-comparison

Fix a misleading method name

funbaker committed to pyvirtobs/pyvo on Oct 13, 2016

solve inconsistency in method naming in strus query analyzer

patrickfrey committed to patrickfrey/strusAnalyzer on Dec 17, 2016

fix method name error

arsy209 committed to nnat425/snowtree on Mar 16

(b) From commit logs in GitHub.

Fig. 2. Excerpts of spotted issues about inconsistent method names.

To debug method name, Høst and Østvold [27] explored method naming rules and semantic profiles of method implementations. Kim *et al.* [7] relied on a custom code dictionary to detect inconsistent names. Allamanis *et al.* introduced the NATURALIZE framework [28] learning the domain-specific naming convention from local contexts to improve the stylistic consistency of code identifiers with n-gram model [29]. Then, building on this framework, they proposed a log-bilinear neural probabilistic language model to suggest method and class names with similar contexts [30]. The researchers leveraged attentional neural networks [31] to extract local time-invariant and long-range topical attention features in a context-dependent way to suggest names for methods.

Overall, their context information is limited to local identifier sub-tokens and the data types of input and output. While the state-of-the-art has achieved promising results, a prime criterion of naming methods has not been considered: the implementation of methods that is a first-class feature to assess method naming consistency since method names should be mere summaries of methods' behavior [1]. Examples shown in Figure 1 illustrate the intuition behind our work:

> *Methods implementing similar behavior in their body code are likely to be consistently named with similar names, and vice versa. It should be possible to suggest new names for a method, in replacement to its inconsistent name, by considering consistent names of similarly implemented methods.*

In this paper, we propose a novel automated approach to spotting and refactoring inconsistent method names. Our approach leverages Paragraph Vector [32] and Convolutional Neural Networks [33] to extract deep representations of method names and bodies, respectively. Then, given a method name, we compute two sets of similar names: the first one corresponds to those that can be identified by the trained model of method names; the second one, on the other hand, includes names of methods whose bodies are positively identified as similar to the body of the input method. If the two sets intersect to some extent (which is tuned by a threshold parameter), the method name is identified to be consistent, and inconsistent otherwise. We further leverage the second set of consistent names to suggest new names when the input method name is flagged as inconsistent.

To evaluate our proposed approach, we perform experiments with 2,116,413 methods of training data and 2,805 methods with changed names of test data, which are collected from 430 open source Java projects. Our experimental results show that the approach can achieve an F1-measure of 67.9% in the identification of inconsistent method names, representing an improvement of about 15 percentage points over the state-of-the-art. Furthermore, the approach achieves 34−50% accuracy on suggesting first sub-tokens and 16−25% accuracy on suggesting accurate full names for inconsistent method names, again outperforming the state-of-the-art. Finally, we report how our approach helped developers in fixing 66 inconsistent method names in 10 projects during a live study in the wild.

II. Background

This section briefly describes three techniques from the field of neural networks, namely Word2Vec [34], Paragraph Vector [32] and Convolutional Neural Networks [33]. Our approach relies on these techniques to achieve two objectives: (1) embedding tokens from method names and bodies into numerical vector forms, and (2) extracting feature representations for accurately identifying similar method names and bodies.

1) Paragraph Vector: Paragraph Vector is an unsupervised algorithm that learns fixed-length feature representations from variable-length pieces of texts, such as sentences and paragraphs [32]. This technique was proposed to overcome the limitations of bag-of-words [35] features which are known to (1) lose the ordering of the words and (2) ignore the word semantics. Recent literature has provided evidence that paragraph vector outperforms other state-of-the-art techniques [35], [36] for text representations [32], [37], and can effectively capture semantic similarities among words and sentences [38]–[42].

In our work, we use Paragraph Vector for training a model to compute similarities among method names (considering sequences of method name sub-tokens as sentences). We expect this model to take into account not only the lexical similarity but also the semantic similarity: for example, function names `containsObject` and `hasObject` should be classified as similar names since both of them describe the functionality of code implementation to check whether a set contains a specific object put in argument(s). We detail in later parts of this paper how method names are processed in our approach to feeding the Paragraph Vector algorithm.

2) Convolutional Neural Networks (CNNs): CNNs are biologically-inspired variants of multi-layer artificial neural networks [33]. Initially developed and proven effectiveness in the area of image recognition, CNNs have gained popularity for handling various NLP tasks. For text classification, in particular, these deep learning models have achieved remarkable results [43], [44] by managing to capture the semantics of sentences for relevant similarity computation. Recently, a number of studies [45]–[51] have provided empirical evidence to support the *naturalness of software* [52], [53]. Thus, inspired by the *naturalness* hypothesis, we treat source code, in particular, method bodies, as documents written in natural language and to which we apply CNNs for code embedding purpose. The objective is to produce a model that will allow to accurately identify similar method code. A recent work by Bui et al. [54] has provided preliminary results showing that some variants of CNNs are even effective to capture code semantics so as to allow the accurate classification of code implementations across programming languages.

In this study, we use LeNet5 [55], a specific implementation of CNNs, which consists of lower-layers and upper-layers (see Figure 4 for its architecture). The lower-layers are composed of alternating convolutional and subsampling layers, which are local-connected. The upper-layers are fully-connected and correspond to a traditional MLP (hidden layer + logistic regression). The input to the first fully-connected layer is the set of all feature maps at the previous layer.

3) Word2Vec: When feeding tokens of a method body to CNNs, it is necessary to convert the tokens into numerical vectors. Otherwise, the size of a CNN's input layer would be too large if using one-hot encoding, or interpreting its output can be distorted if using numeric encoding (i.e., assigning a single integer value for each token). The machine learning community often uses vector representation for word tokens [32], [43], [56]. This offers two advantages: (1) a large number of (unique) tokens can be represented as a fixed-width vector form (dimensionality reduction) and (2) similar tokens can be located in a vector space so that the similar tokens can be dealt with CNNs in a similar way. Our approach uses Word2Vec [34] to embed tokens of method bodies.

Word2Vec [57] is a technique that encodes tokens into n-dimensional vectors [34], [58]. It is basically a two-layered neural network dedicated to process token sequences. The neural network takes a set of token sequences (i.e., sentences) as inputs and produces a map between a token and a numerical vector. The technique not only embeds tokens into numerical vectors but also places semantically similar words in adjacent locations in the vector space.

III. Approach

This section presents our approach to debugging inconsistent method names. As illustrated in Figure 3, it involves two phases: (1) training and (2) identification & suggestion. In the training phase, taking as input a large number of methods from real-world projects, it uses Paragraph Vector for method names and Word2Vec + CNNs for method bodies to embed them into numerical vectors (hereafter simply referred to as vectors), respectively. Eventually, two distinct vector spaces are produced and will be leveraged in the next phase. The objective of the training phase is thus to place similar method names and bodies into adjacent locations in each vector space.

The identification & suggestion phase determines whether a given method has a name that is consistent with its body code by comparing the overlap between the set of method names that are close in the method name vector space and the set of methods names whose bodies are close in the method body vector space. When the overlap is $\emptyset$, the name is considered to be inconsistent with the body code and suggested with alternative consistent names.

Before explaining the details of these two phases, we first describe an essential step of data processing that adapts to the settings of code constructs.

A. Data Preprocessing

This step aims at preparing the raw data of a given method to be fed into the workflow of our approach. We consider the textual representations of code and transform them into tokens (i.e., basic data units) which are suitable for the deep representation learning techniques described in Section II. Given that method names and bodies have different shapes (i.e., names are about natural language descriptions while bodies are focused on code implementations of algorithms), we propose to use tokenization techniques adapted to each:

- **Method name tokenization**: Method names are broken into sub-token sequences based on *camel case* and *underscore* naming conventions, and the obtained sub-tokens are brought to their lowercase form. This strategy has been proven effective in prior studies [7], [27], [28], [30], [31], [59]. For example, method names `findField` and `find_field` are tokenized into the same sequence [`find`, `field`], where `find` and `field` are respectively the first and second sub-tokens of the names.
- **Method body tokenization**: Method bodies are converted into textual token sequences by following the code parsing method proposed in our previous study [60]: this method consists in traversing the abstract syntax tree (AST) of a method body code with a depth-first search algorithm to collect two kinds of tokens: AST node types and raw code tokens. For example, the declaration statement "`int a;`" will be converted into a four-token sequence: [`PrimitiveType`, `int`, `Variable`, `a`]. Since noisy information of code (e.g., non-descriptive variable names such as `a`, `b`) can interfere with identifying similar code [61], all local variables are renamed as the concatenation of their data type with the string `Var`. Eventually, the previous declaration code will be represented by the sequence: [`PrimitiveType`, `int`, `Variable`, `intVar`].

B. Training

This phase takes tokens of method names and bodies in a code corpus to produce two numerical vector spaces that are leveraged to compute similarities, among method names, on the one hand, and among method bodies, on the other hand, for eventually identifying inconsistent names and suggesting appropriate names. Note that the objective is not to train a classifier whose output will be some classification label given a method name or body. Instead, we adopt the idea of unsupervised learning [62] and lazy learning [63] to embed method names and bodies.

Token sequences of method names are embedded into vectors by the paragraph vector technique described in Section II-1 since token sequences of method names resemble sentences describing the methods. In contrast, all tokens in a

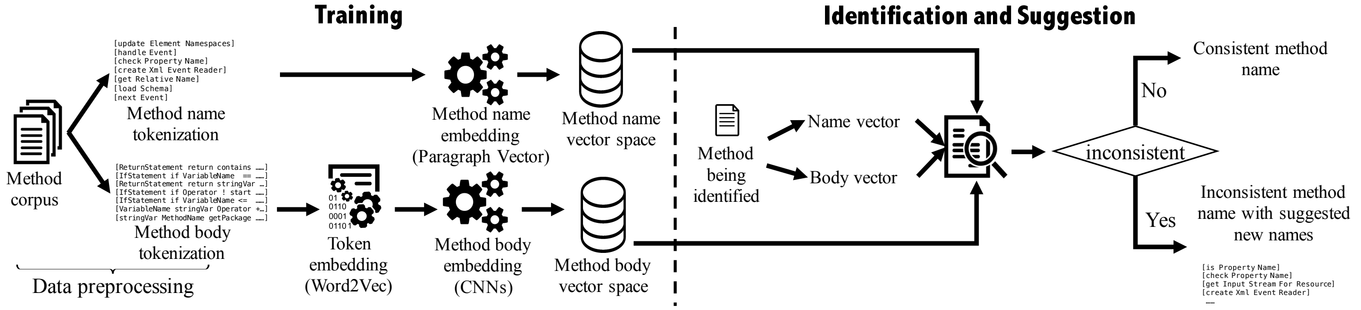

Training
Identification and Suggestion
Method corpus
[update Element Namespaces]
[handle Event]
[check Property Name]
[create Xml Event Reader]
[get Relative Name]
[load Schema]
[next Event]
Method name tokenization
[ReturnStatement return contains ……]
[IfStatement if VariableName == ……]
[ReturnStatement return stringVar …]
[IfStatement if Operator ! start ……]
[IfStatement if VariableName <= ……]
[VariableName stringVar Operator +…]
[stringVar MethodName getPackage ……]
Method body tokenization
Data preprocessing
01 0110 0001 01101
Token embedding (Word2Vec)
Method name embedding (Paragraph Vector)
Method body embedding (CNNs)
Method name vector space
Method body vector space
Method being identified
Name vector
Body vector
inconsistent
No
Consistent method name
Yes
Inconsistent method name with suggested new names
[is Property Name]
[check Property Name]
[get Input Stream For Resource]
[create Xml Event Reader]
……

Fig. 3. Overview of our approach to spotting and refactoring inconsistent method names.

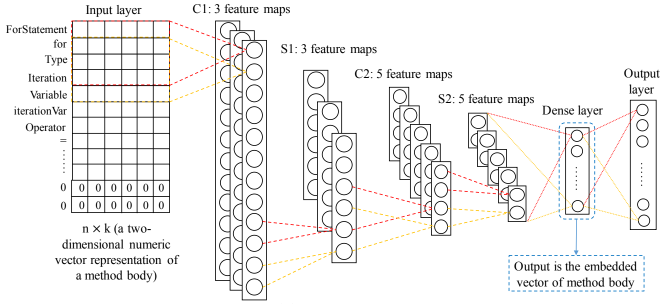


Fig. 4. Architecture of CNNs [55] used in our approach to vectorize method bodies, where C1 and C2 are convolutional layers, and S1 and S2 are subsampling layers, respectively.

method body are first embedded into vectors using Word2Vec. The embedded token vectors are then fed to CNNs to embed the whole method body into a vector, which will be used to represent each method body as a numerical vector.

1) Token Embedding for Method Bodies: As shown in Figure 3, tokens of method bodies are embedded into individual numerical vectors before they can be fed to the CNNs. To that end, the token embedding model is built as below:

$$\mathrm{TV}_B \leftarrow E_W(T_B) \tag{1}$$

where E_W is the token embedding function (i.e., Word2Vec [34] in our case) taking as input a training set of method body token sequences T_B. The output is then a token mapping function $\mathrm{TV}_B : TW_B \rightarrow V_{BW}$, where TW_B is a vocabulary of method body tokens, and V_{BW} is the vector space embedding the tokens in TW_B.

After token embedding, a method body is eventually represented as a two-dimensional numerical vector. Suppose that a given method body b is represented by a sequence of tokens $T_b = (t_1, t_2, t_3, \ldots, t_k)$, where $t_i \in TW_B$, and V_b is a two-dimensional numerical vector corresponding to T_b. Then V_b is inferred as follows:

$$V_b \leftarrow l(T_b, \mathrm{TV}_B) \tag{2}$$

where l is a function that transforms a token sequence of a method body into a two-dimensional numerical vector based on the mapping function TV_B. Thus, $V_b = (v_1, v_2, v_3, \ldots, v_k) \in V_B$, where $v_i \leftarrow \mathrm{TV}_B(t_i)$ and V_B is a set of two-dimensional vectors.

Since token sequences of method bodies may have different lengths (i.e., k could be different for each method body), the corresponding vectors must be padded to comply with a fixed-width input layer in CNNs. Our approach follows the workaround tested by Wang *et al.* [64] and appends PAD vectors (i.e., zero vectors) to make all vector sizes consistent with the size of the longest one (see Section IV-2 for how to determine the longest one). For example, the left side of Figure 4 (See Section III-B2 for its description) shows how a method body is represented by a two-dimensional $n \times k$ numerical vector, where n is the vector size of each token and k is the size of the longest token sequence of bodies. Each row represents a vector of an embedded token, and the last two rows represent the appended zero vectors to make all two-dimensional vector sizes consistent.

2) Embedding Method Names and Bodies into Vectors: Vector spaces are built by embedding method names and bodies into corresponding numerical vectors. For method names, we feed the sub-token sequences (i.e., one sequence per method name) to a paragraph embedding technique. Specifically, we leverage the paragraph vector with distributed memory (PV-DM) technique [32], which embeds token sequences into a vector space as follows:

$$\mathrm{NV}_{name} \leftarrow E_{PV}(T_N) \tag{3}$$

where E_{PV} is the paragraph vector embedding function (i.e., PV-DM), which takes as input a training set of method name sub-token sequences T_N. The output is a name mapping function $\mathrm{NV}_{name} : T_N \rightarrow V_N$, where V_N is an embedded vector space for method names. This step is similar to classical word embedding with differences in the mapping relationships. The paragraph vector embeds a token sequence into a vector, while Word2Vec embeds a token into a vector.

For method bodies, we need another mapping function, where the input is a two-dimensional numerical vector for each method body. The output is a vector corresponding to a body. This mapping function is obtained by the formula below:

$$\mathrm{VV}_{body} \leftarrow E_{BV}(V_B) \tag{4}$$

where E_{BV} is an embedding function (i.e., CNNs) that takes the two-dimensional vectors of method bodies (V_B) as training data and produces a mapping function (VV_{body}). Note that $V_B = \{V_{b_1}, V_{b_2}, V_{b_3}, \ldots, V_{b_m}\}$ is obtained by l (Equation 2), where V_{b_i} ($i \in [1, m]$) and m is the size of training data. VV_{body} is defined as $\mathrm{VV}_{body} : V_B \rightarrow V'_B$, where V'_B is an embedded vector space of method bodies. Based on VV_{body}, we defined the body mapping function NV_{body} as:

$$\mathrm{NV}_{body} : T_B \rightarrow V'_B \tag{5}$$

where NV_{body} is the composition of l and VV_{body} in Equations 2 and 4, respectively (i.e., $\mathrm{NV}_{body} = (VV_{body} \circ l)(T_b) = VV_{body}(l(T_b))$). NV_{body} takes a token sequence of a method body and returns an embedded vector representing it.

Our approach uses CNNs [33] as the embedding function E_{BV} in Equation 5. Figure 4 shows the architecture of CNNs that our approach uses. The input is two-dimensional numeric vectors of method bodies as stated in Section III-B1. The two pairs of convolutional and subsampling layers are used to capture the local features of methods and decrease dimensions of input data. The network layers from the second subsampling layer to the subsequent layers are fully connected, which can combine all local features captured by convolutional and subsampling layers. We select the output of dense layer as the vector representations of method bodies, which synthesizes all local features captured by previous layers.

Note that vectors in the two vector spaces (i.e., V_N and V'_B) can be indexed by each method name. For a given body vector of a method, we can immediately find its corresponding name vector in the name vector space, and vice versa. This index facilitates the search of corresponding method names after locating similar method bodies for a given method.

C. Identification & Suggestion

This phase consists of two sub-steps. First, the approach takes a given method as a query of inconsistency identification. By leveraging the two vector spaces (i.e., V_N and V'_B) and the two embedding functions (i.e., NV$_{name}$ and NV$_{body}$), it identifies whether the name of the given method is consistent with its body. Second, if the name turns out to be inconsistent, the approach suggests potentially consistent names for it from the names of similarly implemented methods.

1) Inconsistency Identification: For a given method m_i, we can take a set of adjacent vectors for its name (n_i) and body (b_i), respectively (i.e., $adj(n_i)$ and $adj(b_i)$). After retrieving the actual names (i.e., $name(*)$) corresponding to vectors in $adj(n_i)$ and $adj(b_i)$, we can compute the intersection between the two name sets as C_{full}:

$$C_{full} = name(adj(n_i)) \cap name(adj(b_i)) \qquad (6)$$

If C_{full} is $\emptyset$, we consider b_i to be inconsistently named n_i.

However, C_{full} in Equation 6 is too strict since it relies on exact matching. In other words, there should exist the same character sequences between two name sets. For example, suppose that there is `findField` in $name(adj(n_i))$ and `findElement` in $name(adj(b_i))$ with similar implementations. This relationship cannot be identified by C_{full} even if they have similar behavior of looking up something.

In the Java naming conventions [65], the first sub-token often indicates the key behavior of a method [66] (e.g., `get[...]()`, `contains[...]()`). Thus, if the key behavior of a given method is similar to those of other methods with similar bodies, we can regard that the name is consistent. Thus, we relax the condition of consistency. Instead of comparing the full name, we take the first sub-token of each name in the two name sets to get the intersection as below:

$$C_{relaxed} = first(name(adj(n_i))) \cap first(name(adj(b_i))) \qquad (7)$$

where $first(*)$ is a function that obtains the first sub-token set by the same rule of method name tokenization described in Section III-A. Other subsequent tokens are often highly project-specific. Therefore, those subsequent tokens would be different across projects even if their bodies are highly similar.

Algorithm 1: Inconsistency identification and new names suggestion.

```
   Input: target method (name and body): $m_i = (n_i, b_i)$
   Input: threshold of adjacent vectors: $k$
   Input: set of name vectors obtained from a training set: $V_N$
   Input: set of body vectors obtained from a training set: $V'_B$
   Input: indexes of actual names from all vectors $\forall V \in V_N$ or $V'_B$:
          Idx$_{name} : V \rightarrow N$
   Input: function embedding a name to a vector: NV$_{name}$
   Input: function embedding a body to a vector: NV$_{body}$
   Output: pair of the consistency determinant of $m_i$ (Boolean) and a set of
           suggested names: $(c, SG_n)$, where $SG_n$ is $\emptyset$ if $c$ is $false$.
1  Function identify($m_i$, $V_N$, $V'_B$)
2      // compute name and body vectors of $m_i$.
3      $V_n$:= NV$_{name}(n_i)$;
4      $V'_b$:= NV$_{body}(b_i)$;
5      // get adjacent name vectors similar to the name vector ($V_n$) of $m_i$.
6      NameV$_{adj}$:= getTopAdjacent($V_n$, $V_N$, $k$);
7      //get actual names for adjacent name vectors (NameV$_{adj}$).
8      Names$^{n_i}_{adj}$:= NameV$_{adj}$.collect(Idx$_{name}$($\forall V \in$ NameV$_{adj}$));
9      // get adjacent body vectors similar to the body vector ($V'_b$) of $m_i$.
10     BodyV$_{adj}$:= getTopAdjacent($V'_b$, $V'_B$, $k$);
11     // get actual names for adjacent body vectors (BodyV$_{adj}$).
12     Names$^{b_i}_{adj}$:= BodyV$_{adj}$.collect(Idx$_{name}$($\forall V \in$ BodyV$_{adj}$));
13     // take the first tokens of actual names for adjacent name and body
         vectors.
14     $fT^{n_i}_{adj}$:= Names$^{n_i}_{adj}$.collect(tokenize$_{name}$($\forall N \in$ Names$^{n_i}_{adj}$).first);
15     $fT^{b_i}_{adj}$:= Names$^{b_i}_{adj}$.collect(tokenize$_{name}$($\forall N \in$ Names$^{b_i}_{adj}$).first);
16     if $fT^{n_i}_{adj} \cap fT^{b_i}_{adj}$ is $\emptyset$ then
17         // $m_i$ has an inconsistent name and suggest new names.
18         newNames:= rankNames(Names$^{b_i}_{adj}$, BodyV$_{adj}$);
19         $(c, SG_n)$:=(false, newNames);
20     else
21         // $m_i$ has a consistent name.
22         $(c, SG_n)$:=(true, $\emptyset$);
```

Algorithm 1 details the precise routine for checking whether the name n_i of a method is consistent with its body b_i or not. Our approach computes the cosine similarity for a given method to search for similar methods. After retrieving the embedded vectors of the name n_i and body b_i (cf. lines 3 and 4), the approach looks up the top k adjacent vectors in the respective vectors spaces for method names and bodies (cf. lines 6 and 10). Since threshold k can affect the performance of identification, our evaluation described in Section V includes an experiment where k values are varied.

After remapping the set of adjacent vectors to sets of the corresponding method names (cf. lines 8 and 12), the sets are processed to keep only first sub-tokens (cf. lines 14 and 15), since our approach uses $C_{relaxed}$ as specified in Equation 7 to compare the two sets of first tokens, $fT^{n_i}_{adj}$ and $fT^{b_i}_{adj}$ (cf. line 16). If their intersection is $\emptyset$, the approach suggests names for the given method body b_i (cf. Section III-C2 for details). Otherwise, our approach assumes that n_i is consistent with b_i.

2) Name Suggestion: Our approach suggests new names for a given method by providing a ranked list of the similar names (cf. line 18), with four ranking strategies as below:

- **R1**: This strategy purely relies on the similarities between method bodies. The names of similar method bodies (Names$^{b_i}_{adj}$) are ranked by the similarities to the given method body (between V'_b and BodyV$_{adj}$).
- **R2**: This strategy first groups the same names in Names$^{b_i}_{adj}$ since there might be duplicates. It then ranks distinct names

based on the size of the associated groups. Ties are broken based on the similarities between method bodies as **R1**.

- **R3**: Similarly to **R2**, this strategy groups the same names in $\text{Names}^{b_i}_{adj}$. Then, the strategy computes the average similarity to b_i of each group and ranks the groups based on the average similarity, but the group sizes are not considered.
- **R4**: To avoid having highly ranked groups with a small size as per strategy **R3**, this strategy eventually re-ranks all groups produced in **R3** by downgrading all 1-size groups to the lowest position.

IV. Experimental Setup

Empirical validation of the approach is performed through various experiments. Before describing the results and conclusions, we present the research questions and the data collection as well as details on the parameter settings in implementation to facilitate replication.

1) Research Questions: To evaluate the approach, we propose to investigate the following research questions (RQs):

- **RQ1**: *How effectively does the approach identify inconsistent method names?*
- **RQ2**: *Can the approach suggest accurate method names?*
- **RQ3**: *How does the approach compare with the state-of-the-art in terms of performance?*
- **RQ4**: *To what extent applying the approach in the wild produces debugging suggestions that are acceptable to developers?*

2) Data Collection: We collect both training and test data from open source projects from four different communities, namely Apache, Spring, Hibernate, and Google. We consider 430 Java projects with at least 100 commits, to ensure that these have been well-maintained.

Training data is constituted by all methods of these projects, after filtering out noisy data with criteria as below:

- *main* methods, constructor methods, and example methods[1] are ignored since they have the less adverse effect on program maintenance and understanding, and can pollute the results of searching for similar methods.
- Empty methods (i.e., abstract or zero-statement methods) have no implementation and thus are filtered out.
- Methods names without alphabetic letters (e.g., some methods are named "_") are removed as they are undescriptive.

As a result, 2,425,939 methods are collected.

In practice, we further limit the training data to methods with reasonable size, to avoid the explosion of code tokens which can degrade performance. The sizes of token sequences of collected method bodies range from 2 to 60,310. According to the sizes' distribution shown in Figure 5, most methods have less than 100 tokens. We focus on building the training data with methods containing at most 94 tokens, which is set based on the upper whisker value[2] from the boxplot distribution of method body token sequence sizes in Figure 5. The sizes beyond the upper whisker value are considered as outliers [67].

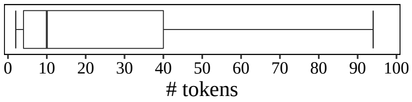


Fig. 5. Sizes' distribution of collected method body token sequences.

TABLE I
Size of datasets used in the experiments.

Classification	# Methods
All collected methods	2,425,939
Methods after filtering (for training)	2,116,413
Methods for testing	2,805

Eventually, 2,116,413 methods are selected to be the training data, as indicated in Table IV-2. Note that methods in the training data are not labeled as *consistent* or *inconsistent* since the objective of training is to construct a vector space of methods with *presumable*[3] consistent names.

Test data is the oracle that we must constitute to assess the performance of our proposed approach. We build it by parsing the commit history of our subjects (i.e., 430 projects). Specifically, we consider:

- Methods whose names have been changed in a commit without any modification being performed on the body code;
- and the names and body code have become stable after the change (i.e., no more changes up to the current versions).

The first criterion allows to ensure that the change is really about fixing method names, and to retrieve their buggy and fixed versions. Overall, within commit changes, we identified 53,731 methods satisfying this criterion. The second criterion increases the confidence that the fixed version of the name is not itself found buggy later on. With this criterion, the number is reduced to 8,734 methods. We further observe that some method names are changed due to simple typos (cf. Figure 6). Such changes can constitute noise in the oracle. Given that our approach heavily relies on first sub-tokens of method names to hint at inconsistency, we conservatively ignore all change cases where this part is not changed. At this stage, the dataset still includes 4,445 buggy-fixed pairs of method names. The final selection follows the criterion used for collecting training data (i.e., no constructor or example methods, etc.). The final test data includes 2,805 distinct methods.

```
Commit 70106770ea61a5fe845653a0b793f4934cc00144
-public double inverseCummulativeProbability(final double p){
+public double inverseCumulativeProbability(final double p){
```

Fig. 6. A typo fix for a method name in Apache commons-math.

To ensure that there is no data leakage [68] between training and test data that will artificially improve the performance of our approach, we eliminate from the training data all methods associated to the test data (i.e., there is no the same instance between 2,116,413 methods in training data and 2,805 methods in test data).

Our test data include method names for each of which we have two versions: the buggy name and the fixed one. To build our oracle, we need two sets, one for the *inconsistent* class and the other for the *consistent* class. We randomly divide our test data into two sets. In the first set, we consider only the buggy versions of the method names and label them as inconsistent.

[1] The package, class or method name includes the keyword "example", "sample" or "template".

[2] The upper whisker value is determined by 1.5 IQR (interquartile ranges) where IQR = 3rd Quartile − 1st Quartile, as defined in [67].

[3] The majority of the methods in the world have names that are likely to be consistent with their bodies.

In the second set, we consider only the fixed versions and label them as consistent.

3) Implementation of Neural Network Models: The Paragraph Vector, Word2Vec, and CNNs models are implemented with the open source DL4J library [69], which is widely used across the research and practice in deep learning (800k+ people and 90k+ communities according to data on the Gitter networking platform [70]). These neural networks must be tuned with specific parameters. In this study, all parameters are set following the parameters setting proposed by Kim [43] and our previous work [60]. Their subjects and models are similar to ours, and their yielded models were shown to achieve promising results. Tables II, III, and IV show the parameters used in our experiment for each model.

TABLE II
PARAMETERS SETTING OF PARAGRAPH VECTOR.

Parameters	**Values**	**Parameters**	**Values**
Min word frequency	1	Size of vector	300
Learning rate	0.025	Window size	2

TABLE III
PARAMETERS SETTING OF WORD2VEC.

Parameters	**Values**	**Parameters**	**Values**
Min word frequency	1	Size of vector	300
Learning rate	1e-2	Window size	4

TABLE IV
PARAMETERS SETTING OF CNNS.

Parameters	**Values**	**Parameters**	**Values**
# nodes in hidden layers	1000	learning rate	1e-2
activation (output layer)	softmax	pooling type	max pool
activation (other layers)	ReLU		
optimization algorithm	stochastic gradient descent		
loss function	mean absolute error		

V. EVALUATION

A. RQ1: Effectiveness of Inconsistency Identification

As the first objective of our approach is to identify *inconsistent* method names, we examine whether our approach effectively identifies methods with inconsistent names. We train the model with the collected training data and apply it to the separated test data whose collection was described in Section IV-2. Given that the performance of identification depends on the threshold value k representing the size of the sets of adjacent vectors (Lines 6 and 10 in Algorithm 1), we vary k as 1, 5, and $n \times 10$ with $n \in [1, 10]$.

Inconsistent identification is a binary classification since the test data explained in Section IV-2 are labeled in two classes (IC: inconsistent, C: consistent). Thus, there are four possible outcomes: IC classified as IC (i.e, true positive=TP), IC classified as C (i.e., false negative=FN), C classified as C (i.e., true negative=TN), and C classified as IC (i.e., false positive=FP). We compute Precision, Recall, F1-measure and Accuracy for each class. The precision and recall for the class IC are defined as $\frac{|TP|}{|TP|+|FP|}$ and $\frac{|TP|}{|TP|+|FN|}$, respectively. Those for the class C are defined as $\frac{|TN|}{|TN|+|FN|}$ and $\frac{|TN|}{|TN|+|FP|}$, respectively. The F1-measure of each class is defined as $2 \cdot \frac{precision \cdot recall}{precision+recall}$, while the Accuracy is defined as $\frac{|TP|+|TN|}{|TP|+|FP|+|TN|+|FN|}$.

Table V provides the experimental results on the performance. Due to space limitation, we show the metrics for variations of k up to 40 (instead of 100). Overall, our approach yields an Accuracy metric ranging from 50.8% to 60.9% (for the presented results) and an F1-measure up to 67.9% for the inconsistent class. In particular, The approach achieves the highest performance when k=1 (i.e., the single most adjacent vector is considered). The general trend indeed is that the performance decreases as k is increased.

TABLE V
EVALUATION RESULTS OF INCONSISTENCY IDENTIFICATION.

	Evaluation metrics	k = 1	k = 5	k = 10	k = 20	k = 30	k = 40
Inconsistent	Precision (%)	**56.8**	53.7	53.3	53.3	49.9	49.7
	Recall (%)	**84.5**	55.9	46.7	46.7	28.8	33.6
	F1-measure (%)	**67.9**	54.8	49.7	49.7	36.5	40.1
Consistent	Precision (%)	**72.0**	55.9	54.2	54.2	51.4	51.4
	Recall (%)	38.2	53.7	60.7	60.7	**72.2**	67.4
	F1-measure (%)	49.9	54.8	57.3	57.3	**60.0**	58.3
Accuracy (%)		**60.9**	54.8	53.8	50.8	50.9	51.1

Since k determines the number of similar methods retrieving from the vector spaces of names and bodies in the training data, higher k value increases the probability of non-empty intersection (i.e., $\text{fT}^{n_i}_{adj} \cap \text{fT}^{b_i}_{adj}$, cf. line 16 in Algorithm 1). Thus, the recall of the *inconsistent* class tends to decrease as k is getting higher. In contrast, the recall of the *consistent* class increases for higher values of k.

Overall, the approach to identifying inconsistent method names can be tuned to meet the practitioners' requirements. When the criteria are to identify as many inconsistent names as possibles, k should be set to a low value.

B. RQ2: Accuracy in Method Names Suggestion

This experiment aims at evaluating the performance of our approach in suggesting new names for identified inconsistent names. The suggested names are ranked by a specifiable ranking strategy (either R1, R2, R3, or R4 as described in Section III-C2).

Prior studies compare the first tokens of suggested names [27], [71], [72] and oracles or token sets without considering token ordering [31], [73]. To ensure fair and comprehensive assessment, we consider three different scenarios to evaluate the performance of our approach. These scenarios are defined as follows:

- **T1** (Inconsistency avoidance): In this scenario, we evaluate to what extent the suggested names are different from the input buggy name n_i. The accuracy in this scenario is computed by $1 - \frac{\Sigma_{i \in ic} D\left(n_i^1, ft(\text{BodyV}_{adj})\right)}{|ic|}$, where n_i^1 is the first token of n_i. $ft()$ collects the first tokens of method names corresponding to each vector in BodyV_{adj} (similar body vectors shown at Line 10 in Algorithm 1). $D(*)$ checks whether the items in the second argument contain the first argument (if so, returns 1. Otherwise 0). ic is the set of inconsistent method names identified by our approach.
- **T2** (First-token accuracy): In this scenario, we evaluate to what extent the suggested names are identical to the name that developers proposed in debugging changes. We remind the reader that our test data indeed include pairs

of buggy/fixed names extracted from code changes history (cf. Section IV-2). The accuracy is computed as $\frac{\Sigma_{i \in ic} D\big(rn(n_i)^1, ft(\text{BodyV}_{adj})\big)}{|ic|}$, where $rn(n_i)^1$ is the first token of the actual developer-fixed version of the name.

- **T3** (Full-name accuracy): In this scenario, we evaluate to what extent the full name of each suggested name is identical to the name that developers proposed in debugging changes. The accuracy in this scenario is computed as $\frac{\Sigma_{i \in ic} D\big(rn(n_i), fn(\text{BodyV}_{adj})\big)}{|ic|}$, where $rn(n_i)$ is the actual fixed version of the name and $fn(*)$ retrieves the full names of methods corresponding to each vector in BodyV_{adj}.

We perform different experiments, varying k. For these experiments, the approach produces new names for all methods identified as *inconsistent* (i.e., true_positive + false_positive). Thus, the results include the performance of false positives (i.e., names that are already consistent). We compute the performance based on the number of suggested names (varying the threshold value thr). For the ranking strategy R1, k is set only to 1 or 5 since higher values do not affect the results when $thr = 1$ or 5 (note that R1 produces the same number of suggested names with k). For other ranking strategies, k is set to 10, 20, 30, and 40 to have large numbers of suggested names that can be aggregated (as per ranking strategy working).

According to the results for **T1** listed in Table VI, our approach is highly likely to suggest names that are different from the identified inconsistent names with ranking strategies, even when k is high (>90% if thr=1 and >60% if thr=5). When matching the first tokens (cf. results for **T2**) and the full name (cf. results for **T3**), the ranking strategy R4 slightly outperforms others regardless of the k value, while its accuracy is ≈40%. thr=5 gives a better probability to find consistent names than thr=1. k=10 yields the best performance for ranking strategies R2 and R3 while ranking strategy R4 performs best when k=20 and thr=5 and k=40 and thr=1.

TABLE VI
ACCURACY OF SUGGESTING METHOD NAMES WITH THE FOUR RANKING STRATEGIES (I.E., R1, R2, R3 AND R4).

Accuracy		k = thr	k = 10			k = 20			k = 30			k = 40		
(%)		R1	R2	R3	R4	R2	R3	R4	R2	R3	R4	R2	R3	R4
T1	thr=1	90.0	91.2	91.3	**76.1**	92.2	92.2	75.9	93.0	92.9	75.8	**93.7**	**93.6**	75.9
	thr=5	69.0	**66.4**	**66.4**	**66.1**	64.1	64.0	61.8	64.9	64.9	61.3	66.1	66.1	61.5
T2	thr=1	23.4	**23.2**	**23.0**	24.1	21.5	21.5	24.1	19.3	19.3	24.0	17.2	17.2	**24.2**
	thr=5	35.7	**39.4**	**39.4**	39.7	38.5	38.6	**40.8**	37.3	37.2	40.6	36.5	36.3	40.1
T3	thr=1	10.7	**11.0**	**10.9**	10.9	10.9	10.9	11.1	10.6	10.5	11.3	10.2	10.1	**11.5**
	thr=5	17.0	**18.7**	**19.0**	19.2	17.7	17.8	**19.5**	16.9	16.9	19.4	16.6	16.6	19.2

The best case of each ranking strategy in each row is highlighted as **bold**.

The results above show that: (1) ranking strategies R2, R3, and R4 perform better than R1 but they need more candidates, (2) higher k values do not increase the accuracy of name suggestion, and (3) more number of suggested names (i.e., higher thr values) would improve the accuracy but users of our approach will need to look up more names.

Note that it is promising that achieving ≈20% and ≈40% accuracy (for first-token / T2) when looking up only top-1 and top-5 suggestions, respectively. Suggesting exact first tokens of method names is challenging since there are a large number of available words for the first tokens of method names. Finding the exact full name of a method is even more challenging since full method names are often very project-specific [52]. Our approach achieves ≈10% and ≈20% accuracy, respectively for thr=1 and thr=5.

C. RQ3: Comparison Against the State-of-the-art Techniques

We compare our approach with two state-of-the-art approaches in the literature which are based on the n-gram model [74] and the convolutional attention network (CAN) model [31]. The latter 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). These techniques are selected since they are the most recent approaches for method name debugging. Given that the n-gram model approach by Suzuki et al. [74] cannot suggest full names or even the first token of methods, we compare against them with respect to the performance of inconsistent name identification. The CAN model, on the other hand, does not explicitly identify name inconsistency. Instead, the model suggests names for any given method. Thus, in this experiment, we make the CAN model and our approach suggest names for all test data (2,805 buggy method names). For both techniques, we use the same training data described in Section IV-2. It should be noted that while the tool for the CAN model has been made available by the authors, we had to replicate the n-gram models approach, in a best effort way following the details available in [74].

Table VII shows the comparison results with the n-gram model [74]. While the performance of the n-gram model stays in a range from 51.5-54.2% for all measures, our approach outperforms the model when k=1 and 5. With k=1, the improvement is up to 33 percentage points. In particular, our approach achieves a higher F1-measure by 15 percentage points.

TABLE VII
COMPARISON RESULTS OF IDENTIFYING INCONSISTENT METHOD NAMES AGAINST THE N-GRAM MODEL [74].

Evaluation Metrics	Our Approach			n-gram Model
	k = 1	k = 5	k = 10	
Precision	**56.8%**	53.7%	53.3%	53.3%
Recall	**84.5%**	55.9%	46.7%	51.5%
F1-measure	**67.9%**	54.8%	49.7%	52.4%
Accuracy	**60.9%**	54.8%	53.8%	54.2%

To compare our approach against the CAN model [31], we propose two evaluations. The first follows the evaluation strategy proposed by the authors themselves in their paper. The second evaluation is based on our own strategies already explored for RQ2 (cf. Section V-B).

Table VIII shows the performance based on the *per-sub-token basis* metric, which is the evaluation metric used by the authors originally to present the performance of the CAN model [31]. This metric estimates to what extent sub-tokens of method names can be correctly suggested without considering their order within the method names. We compute precision, recall, and F1-measure of correctly suggesting sub-tokens.

When applying the per-sub-token basis, our approach outperforms the CAN model in all configurations, except for the precision of *copy_attention* with thr=5 (cf. Section V-B). While the precision of our approach can be higher by up

TABLE VIII
COMPARISON OF THE CAN MODEL [31] AND OUR APPROACH BASED ON THE PER-SUB-TOKEN CRITERION [31].

	Precision		**Recall**		**F1-meansure**	
	thr = 1	thr = 5	thr = 1	thr = 5	thr = 1	thr = 5
conv_attention	23.2%	36.5%	8.1%	13.1%	11.7%	18.7%
copy_attention	28.4%	**67.0%**	10.0%	27.5%	14.4%	37.9%
R1 (k = thr)	29.7%	38.6%	27.4%	36.7%	28.5%	37.6%
R2 (k = 10)	30.1%	39.6%	**27.6%**	37.2%	**28.8%**	38.3%
R3 (k = 10)	**30.2%**	39.9%	**27.6%**	**37.6%**	**28.8%**	**38.7%**
R4 (k = 10)	27.2%	38.6%	25.2%	**37.6%**	26.2%	38.1%

to 7 percentage points, we achieve substantial performance improvement in terms of recall and F1-measure, with up to 15 percentage points margin.

Table IX presents the comparison results when applying the three evaluation strategies of RQ2, described in Section V-B. Regardless of the evaluation strategy, our approach outperforms the CAN models. Notably, our approach achieves **16∼25%** accuracy for **T3** (i.e., full name suggestion) while the CAN model is only successful for at most **1.1%**. Note that specific values of accuracy in Table IX are different from Table VI since, in the experiment for RQ3, our approach suggests names for all test data.

TABLE IX
COMPARISON OF THE CAN MODEL [31] AND OUR APPROACH BASED ON THREE EVALUATION SCENARIOS.

Accuracy	**T1**		**T2**		**T3**	
	thr = 1	thr = 5	thr = 1	thr = 5	thr = 1	thr = 5
conv_attention	78.4%	27.6%	22.3%	33.6%	0.3%	0.6%
copy_attention	77.2%	38.9%	23.5%	44.7%	0.4%	1.1%
R1 (k = thr)	86.9%	**69.7%**	36.4%	47.2%	16.5%	22.9%
R2 (k = 10)	88.5%	67.5%	34.8%	50.2%	**17.0%**	25.4%
R3 (k = 10)	**88.6%**	67.5%	34.7%	50.3%	16.9%	25.5%
R4 (k = 10)	77.0%	67.3%	**35.4%**	**50.5%**	16.0%	**25.7%**

While state-of-the-art techniques directly train a classifier for identification or a neural network for suggestion by using a set of training data, our approach first transforms method names and bodies into vectors by using neural networks and then searches for similar vectors by computing distances between them. In that sense, our approach is implemented based on unsupervised learning. Overall, the results imply that looking up similar methods in vector spaces is more effective both for identification and suggestion than other techniques.

D. RQ4: Live Study

To investigate practicability of our approach to debugging inconsistent method names (**RQ4**), we conduct a live study on active software projects: we submit pull requests of renaming suggestions from our approach, and assess acceptance rates. For this experiment, we randomly sample 10% of the training data to be used as test data. Indeed the labeled test data collected for previous experiments represent cases where developers debugged the method names. The remaining 90% of method names now constitute the training data for this phase. We apply this version of our approach to the target subjects to identify whether they have inconsistent names (using k=20 as per result of previous experiments). Overall, 4,430 methods among the 211,642 methods in the test set have been identified as inconsistent by our approach. Given that we cannot afford to spam project maintainers with thousands of pull requests, we randomly select 100 cases of identified inconsistent methods. We then collect the ranked list of suggested names for each of the 100 methods: we use thr=5 with ranking strategy R4 since these parameters show the best performance for full name suggestion (T3). From each ranked list of suggested names, we select the top-1 name and prepare a patch that we submit as a pull request to the relevant project repository.

TABLE X
RESULTS OF LIVE STUDY.

Agree			**Agree but not fixed**		**Disagree**	**Ignored**	**Total**
Merged	Approved	Improved	Cannot	Won't			
40	26	4	1	2	9	18	100

As listed in Table X, developers agreed to merge the pull requests for renaming 40 out of the 100 methods. 26 renaming suggestions have been validated and approved (based on developers' reply) by developers, but the pull requests have not been merged (as of submission date) since some projects systematically apply unit test and complete review tasks of external changes before accepting them into the main branch. Four inconsistent method names have also been fixed after improving our suggested names. Interestingly, one developer used our suggestion as a renaming pattern to fix six (6) similar cases other than the ones submitted in our pull requests. Furthermore, some developers have welcomed our suggestions on inconsistent method names and showed interest in applying even more suggestions from our approach, given that it seems to provide more meaningful names than their current names. Note that 39 (56%) out of 70 names agreed by developers are for *public* methods while remaining eight and 23 are for *private* and *protected* methods.

We also report on cases where developers did not apply our suggested name changes. In one case, the developers *could not* merge the pull request as it would break the program: the method is actually an overridden method from another project. The developer nevertheless agreed that our suggestion was more intuitive. For other two methods, developers agree that the suggested names are appropriate but they *would not* make the changes as the names are not in line with inner-project naming conventions. For nine methods, however, the pull requests are rejected since developers judge the original method names to be more meaningful than the suggested ones. The remaining 18 cases are simply *ignored*: we did not receive any reply up to the date of submission. We summarize developers' feedback as follows:

1) Some method names should follow the naming convention of specific projects. This is a threat to the validity of our study since it is implemented in a cross-project context.
2) Some method should be named considering the class names. E.g., in a class named "XXXBuilder", the developers do not want to name a method as "build", although the method builds a new "XXXBuilder" object.

VI. DISCUSSION

1) Naming based on Syntactic and Semantic Information: As stated in Section I, our approach is based on the assumption that similar method implementations might be associated with similar method names. However, there could be several different definitions of similarity. While our approach relies on the syntactic similarity of method bodies (although, using AST tokens), one can use dynamic information (e.g., execution traces) to compare different method implementations as experimented for the detection of semantic (i.e., type-4) code clones [75], [76]. However, obtaining dynamic information is not scalable. Although we can leverage code-to-code search techniques [77], it might not precise enough for inconsistent name detection. Thus, we leverage only static and syntactic information in our approach and rely on deep learning representations that have been shown to be effective capturing semantics even for code [54], [73].

2) Threats to Validity: A threat to external validity is in the training data since it is impossible to absolutely ensure that all methods in training data have consistent names. To address this threat, we collect training data from the well-maintained open source projects with high reputation. Although the number of projects may not be representative of the whole universe, it is the largest dataset used in published literature about debugging method names. Our live study further demonstrates that the training set is sufficient to build a good model. Another threat to external validity is the typos and abbreviations in method names that can noise method name embedding and suggestion.

Threats to internal validity include the limitation of parsing method names since some method names are named without following camel case or underscore naming convention. It is challenging to parse this kind of method names. This threat could be reduced by developing more advanced method name parse tools with natural language processing. Another threat to validity is the size of data set for testing since the test data is no less than 10% for evaluation in recent machine learning and natural language process literature, where the training and test data are split from collected data. In our study, test data must be actual fixed method names to evaluate the performance of debugging inconsistent method names, but projects used for training do not have such a high number of fixed method names to satisfy the requirement of the balanced training and test data.

VII. Related Work

There have been several empirical studies [66], [78]–[80] investigating the impact of a naming scheme on program comprehension, readability, and maintainability. Takang *et al.* [12] and Lawrie *et al.* [14] conducted empirical studies on code identifiers and concluded that inconsistent names can make source code harder to understand and maintain. Caprile and Tonella [81] analyzed function identifiers from the lexical, syntactical and semantic structure, and reported that identifiers can be decomposed into fragments and further classified into several lexical categories. Liblit *et al.* [13] examined how human cognition is reflected in naming things of programs.

Several approaches have been presented to detect inconsistent identifiers. Deissenboeck and Pizka [8], and Lawrie *et al.* [82] relied on the manual mapping between names and domain concepts to detect inconsistent identifiers in code. Binkley *et al.* [72] developed a tool with part-of-speech tagging to identify field identifiers that violate accepted patterns.

The ultimate goal of debugging names is to automatically replace inconsistent names into consistent ones rather than just helping identifier naming. Haiduc *et al.* [83] used natural language summarization techniques and the lexical and structural context in code to improve code comprehension [84]. Sridhara *et al.* [85] designed an automatic technique for summarizing code with the idioms and structure in a method. Lucia *et al.* [86] proposed an IR-based approach to improve program comprehension with the textual similarity between the code under development and related artefacts. Høst and Østvold [27] used method naming rules and semantic profiles of method implementations to debug method names.

Recently, Allamanis *et al.* [30], [31] leveraged deep learning techniques to suggest method names with local contexts, which are similar to this paper on embedding method names and bodies. Their work learns method body features from code sub-tokens, this paper further consider code nodes at abstract syntax tree level since they can capture code semantic information [47]. This paper performed various evaluations on the actual fixed method names which were not done in [30], [31], with a large sample of 430 projects against the 20/10 projects in their work. In addition, this paper tried various configurations and strategies and used various indicators for method name suggestions, which was not exactly the same as they did. Thus, our conclusions are likely to be more solid than those in their work. Furthermore, we performed a live study (not done in their work) and showed the technique has strong potential to be useful by actually fixing 66 inconsistent method names in the wild.

VIII. Conclusion

Method names are key to readable and maintainable code, but it is not an easy task to give an appropriate name to a method. Thus, many methods have inconsistent names, which can impede the readability and maintainability of programs and even lead to some defects. To reduce the manual efforts of resolving inconsistent method names, we propose a novel approach to debugging inconsistent method names by leveraging similar methods with deep learning techniques. Our experimental results show that the performance of our approach achieves an F1-measure of 67.9% on identifying inconsistent method names, improving about 15 percentage points over the state-of-the-art. On suggesting appropriate first sub-tokens and full names for inconsistent method names, it achieves 34−50% and 16−25% accuracy respectively, outperforming the state-of-the-art as well. We further report that our approach helps developers to fix 66 inconsistent method names in the wild. The tool and data used in our study are available at https://github.com/SerVal-DTF/debug-method-name.

Acknowledgements

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects RECOMMEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[2] S. McConnell, *Code complete*. Pearson Education, 2004.

[3] K. Beck, *Implementation patterns*. Pearson Education, 2007.

[4] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[5] P. Johnson, "Don't go into programming if you don't have a good thesaurus," https://www.itworld.com/article/2833265/cloud-computing/don-t-go-into-programming-if-you-don-t-have-a-good-thesaurus.html, Last Accessed: August 2018.

[6] ——, "Arg! the 9 hardest things programmers have to do," http://www.itworld.com/article/2823759/enterprise-software/124383-Arg-The-9-hardest-things-programmers-have-to-do.html#slide10, Last Accessed: August 2018.

[7] S. Kim and D. Kim, "Automatic identifier inconsistency detection using code dictionary," *Empirical Software Engineering*, vol. 21, no. 2, pp. 565–604, 2016.

[8] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

[9] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "CodeTopics: which topic am i coding now?" in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1034–1036.

[10] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2014.

[11] F. Deissenboeck and M. Pizka, "Concise and consistent naming: ten years later," in *Proceedings of the 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 3–3.

[12] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experimental investigation," *J. Prog. Lang.*, vol. 4, no. 3, pp. 143–167, 1996.

[13] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group*. Citeseer, 2006, pp. 53–67.

[14] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proceedings of the 14th International Conference on Program Comprehension*. IEEE, 2006, pp. 3–12.

[15] V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc, "Repent: Analyzing the nature of identifier renamings," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 502–532, 2014.

[16] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[17] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[18] J. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," in *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2017, pp. 217–227.

[19] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *Proceedings of 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 31–35.

[20] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *Proceedings of the 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2011, pp. 125–134.

[21] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc, "Can lexicon bad smells improve fault prediction?" in *Proceedings of the 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 235–244.

[22] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of api-misuse detectors," *arXiv preprint arXiv:1712.00242*, 2017.

[23] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[24] Eclipse, "Aspectj," https://github.com/eclipse/org.aspectj, Last Access: August. 2018.

[25] S. Exchange, "Stack overflow," https://stackoverflow.com/, Last Access: August. 2018.

[26] Microsoft, "Github," https://github.com/, Last Access: August. 2018.

[27] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proceedings of the 23rd European Conference on Object-Oriented Programming*. Springer, 2009, pp. 294–317.

[28] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 281–293.

[29] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational Linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[30] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[31] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning*. JMLR.org, 2016, pp. 2091–2100.

[32] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning*. JMLR.org, 2014, pp. 1188–1196.

[33] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda, "Subject independent facial expression recognition with robust face detection using a convolutional neural network," *Neural Networks*, vol. 16, no. 5-6, pp. 555–559, 2003.

[34] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[35] G. E. Dahl, R. P. Adams, and H. Larochelle, "Training restricted boltzmann machines on word observations," *arXiv preprint arXiv:1202.5695*, 2012.

[36] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. ACL, 2015, pp. 1422–1432.

[37] Q. Ai, L. Yang, J. Guo, and W. B. Croft, "Analysis of the paragraph vector model for information retrieval," in *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval*. ACM, 2016, pp. 133–142.

[38] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *Proceedings of the 32nd International Conference on Machine Learning*. JMLR.org, 2015, pp. 957–966.

[39] J. Wieting, M. Bansal, K. Gimpel, and K. Livescu, "Towards universal paraphrastic sentence embeddings," *arXiv preprint arXiv:1511.08198*, 2015.

[40] A. M. Dai, C. Olah, and Q. V. Le, "Document embedding with paragraph vectors," *arXiv preprint arXiv:1507.07998*, 2015.

[41] A. Kumar, O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, V. Zhong, R. Paulus, and R. Socher, "Ask me anything: Dynamic memory networks for natural language processing," in *Proceedings of the 33nd International Conference on Machine Learning*. JMLR.org, 2016, pp. 1378–1387.

[42] D. Tang, B. Qin, and T. Liu, "Learning semantic representations of users and products for document level sentiment classification," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, vol. 1. ACL, 2015, pp. 1014–1023.

[43] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. ACL, 2014, pp. 1746–1751.

[44] P. Wang, J. Xu, B. Xu, C. Liu, H. Zhang, F. Wang, and H. Hao, "Semantic clustering and convolutional neural network for short text categorization," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, vol. 2, 2015, pp. 352–357.

[45] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management*. Springer, 2015, pp. 547–553.

[46] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proceedings of the 32nd International Conference on Machine Learning*. JMLR.org, 2015, pp. 2123–2132.

[47] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI, 2016, pp. 1287–1293.

[48] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[49] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 135–146.

[50] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE/ACM, 2017, pp. 438–449.

[51] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 933–944.

[52] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 2012, pp. 837–847.

[53] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, p. 81, 2018.

[54] N. D. Q. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," in *Proceedings of the Workshops of the The 32nd AAAI Conference on Artificial Intelligence*. AAAI Press, 2018, pp. 758–761.

[55] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[56] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," pp. 1724–1734, 2014.

[57] Google, "Word2vec," https://code.google.com/archive/p/word2vec/, Last Accessed: August. 2018.

[58] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems*. NIPS, 2013, pp. 3111–3119.

[59] E. W. Høst and B. M. Østvold, "The java programmer's phrase book," in *Proceedings of the First International Conference on Software Language Engineering*. Springer, 2008, pp. 322–341.

[60] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, 2018.

[61] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "AntMiner: mining more bugs by reducing noise interference," in *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*. ACM, 2016, pp. 333–344.

[62] T. Hastie, R. Tibshirani, and J. Friedman, "Unsupervised learning," in *The Elements of Statistical Learning*. Springer, 2009, pp. 485–585.

[63] D. W. Aha, *Lazy learning*. Washington, DC: Springer, 1997.

[64] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

[65] Oracle, "Java naming convention," http://www.oracle.com/technetwork/java/codeconventions-135099.html, Last Access: August. 2018.

[66] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Mining java class naming conventions," in *Proceedings of the 27th IEEE International Conference on Software Maintenance*. IEEE, 2011, pp. 93–102.

[67] M. Frigge, D. C. Hoaglin, and B. Iglewicz, "Some implementations of the boxplot," *The American Statistician*, vol. 43, no. 1, pp. 50–54, 1989.

[68] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you?: Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 1135–1144.

[69] Eclipse, "Deep learning for java," https://deeplearning4j.org/, Last Access: August. 2018.

[70] Gitter, "Deeplearning4j communities," https://gitter.im/deeplearning4j/deeplearning4j, Last Access: August. 2018.

[71] A. Thies and C. Roth, "Recommending rename refactorings," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 2010, pp. 1–5.

[72] D. Binkley, M. Hearn, and D. Lawrie, "Improving identifier informativeness using part of speech information," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 203–206.

[73] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proceedings of the 46th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, vol. 3. ACM, 2019, pp. 40:1–40:29.

[74] T. Suzuki, K. Sakamoto, F. Ishikawa, and S. Honiden, "An approach for evaluating and suggesting method names using n-gram models," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 271–274.

[75] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 301–310.

[76] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: detecting similarly behaving software," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 702–714.

[77] K. Kim, D. Kim, T. F. Bissyande, E. Choi, L. Li, J. Klein, and Y. Le Traon, "Facoy–a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018.

[78] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 156–165.

[79] S. Butler, "Mining java class identifier naming conventions," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 2012, pp. 1641–1643.

[80] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "INVocD: identifier name vocabulary dataset," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE, 2013, pp. 405–408.

[81] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proceedings of the 6th Working Conference on Reverse Engineering*. IEEE, 1999, pp. 112–122.

[82] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2006, pp. 139–148.

[83] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

[84] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 223–226.

[85] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 101–110.

[86] A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 205–227, 2011.