A Context-based Automated Approach for Method Name Consistency Checking and Suggestion

Yi Li
Department of Informatics
New Jersey Institute of Technology
New Jersey, USA
Email: yl622@njit.edu

Shaohua Wang*
Department of Informatics
New Jersey Institute of Technology
New Jersey, USA
Email: davidsw@njit.edu

Tien N. Nguyen
Computer Science Department
The University of Texas at Dallas
Texas, USA
Email: tien.n.nguyen@utdallas.edu

2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00060

***Abstract*—Misleading method names in software projects can confuse developers, which may lead to software defects and affect code understandability. In this paper, we present DEEPNAME, a context-based, deep learning approach to detect method name inconsistencies and suggest a proper name for a method. The key departure point is the philosophy of *"Show Me Your Friends, I'll Tell You Who You Are"*. Unlike the state-of-the-art approaches, in addition to the method's body, we also consider the *interactions* of the current method under study with the other ones including the caller and callee methods, and the sibling methods in the same enclosing class. The sequences of sub-tokens in the program entities' names in the contexts are extracted and used as the input for an RNN-based encoder-decoder to produce the representations for the current method. We modify that RNN model to integrate the copy mechanism and our newly developed component, called the *non-copy mechanism*, to emphasize on the possibility of a certain sub-token *not to be copied to follow* the current sub-token in the currently generated method name.**

We conducted several experiments to evaluate DEEPNAME on large datasets with +14M methods. For consistency checking, DEEPNAME improves the state-of-the-art approach by 2.1%, 19.6%, and 11.9% relatively in recall, precision, and F-score, respectively. For name suggestion, DEEPNAME improves relatively over the state-of-the-art approaches in precision (1.8%–30.5%), recall (8.8%–46.1%), and F-score (5.2%–38.2%). To assess DEEPNAME's usefulness, we detected inconsistent methods and suggested new method names in active projects. Among 50 pull requests, 12 were merged into the main branch. In total, in 30/50 cases, the team members agree that our suggested method names are more meaningful than the current names.

***Index Terms*—Naturalness of Software, Deep Learning, Entity Name Suggestion, Inconsistent Method Name Checking.**

I. INTRODUCTION

Meaningful and succinct names of program entities play a vital role in code understandability [3]. Misleading names in software libraries can confuse developers and cause them make API misuses, leading to serious defects [10]. During software development, the name of a method can become inconsistent with respect to its intended functionality. The first scenario is that the inconsistency occurs during the coding of a method, when a misleading or confusing name is given to the method. In the second scenario, inconsistency occurs during software evolution in which code changes make the method's name and its implementation become inconsistent with one another.

* Corresponding Author

Several approaches were proposed to detect the inconsistency between the methods' names and source code, and to suggest an alternative name if such inconsistency occurs. The approaches follow mainly two directions: *information retrieval* (IR) [18], [20] and *machine learning* (ML) [3], [4], [7], [28]. The idea of IR approaches is that similar methods should have similar names [20]. Thus, they search for the names of methods with similar bodies to suggest for a method with an inconsistent name. The IR approaches generally follow a searching strategy, thus, cannot recommend a new method name that is un-seen in the training data. The second direction is machine learning (ML) [3], [4], [7], [28]. The ML approaches can overcome the key limitation of the IR direction due to its capability of generating a new name. While *code2vec* [7] generates the method's name based on the paths over the AST of its body, MNire [28] uses the sub-tokens in the program entities' names. Other approaches treat the method name suggestion problem as the extreme summarization [4] from the method's body into a short text. Despite their successes, the state-of-the-art ML approaches have limitations in dealing with the methods having little content or the entities' names that are irrelevant to the functionality.

In this paper, in addition to using the body and interface of the method under study, we also leverage a philosophy for this naming problem: *"Show Me Your Friends, I'll Tell You Who You Are"*. That is, to characterize an entity/person, in addition to using its/his/her own properties, one can rely on the interactions of that entity/person with the surrounding and neighboring entities/persons. For the method name suggestion, examining only the content of the current method might be insufficient. The surrounding and interacting methods of a method m under study could include the methods that are called within the body of m (*callees*), and the methods that are calling m (*callers*). The neighboring methods are the ones within the same class with m (*siblings*). The information from the *enclosing class* also provides features for such characterization of a method. The key features from the *caller* and *callee* methods, *sibling* methods, and the *enclosing class* are used in addition to the features from the *internal body* and *interface* of the method m to verify the consistency of the method with regard to its name, and to suggest a proper method name. Each of those sources constitutes a context that is helpful

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00060

for method name consistency checking and recommendation. Some methods have little content, but with sufficient contexts and vice versa. Thus, all contexts are complementary to one another in name consistency checking and suggestion.

We develop DEEPNAME, a context-based approach for method name consistency checking and suggestion. For the method m under study, it extracts the features from four contexts: the *internal context*, the *caller and callee contexts*, *sibling context*, and *enclosing context*. For name suggestion, only the callee methods are used because the callers of m might not be written yet (the current method does not have a name yet). We use the name features, specifically, the sequences of sub-tokens from the program entities within each context, instead of AST or PDG. It is reported that to infer a method name, using sub-tokens yields better accuracy than using the AST and PDG of the method [28]. The insight is that the naturalness of names plays crucial role in method name inference, i.e., method name depends more on entities' names than AST or PDG (with data/control flows) [28]. AST and PDG capture the structure and procedure of the task, while the method's name is the summary of the task. DEEPNAME uses an RNN-based encoder-decoder to combine all the sequences of sub-tokens in the contexts into a sequence of vectors for m. A convolution layer is used on the vector for m to classify the given name to be consistent or not. To suggest a name for the given method, we use our vocabulary to map all the generated vectors into the sub-tokens to compose the method name.

We also modify the operations of the aforementioned RNN-based encoder-decoder to integrate the *copy* mechanism [11] and the novel *non-copy mechanism*. A recent study on the methods' names [28] has reported that the high percentage of the sub-tokens of a method name appears in a set of the sub-tokens from entities' names in a method. Due to this, the copy mechanism helps emphasize on the possibility of copying certain sub-tokens from the contexts to the output, i.e, the suggested method name. The non-copy mechanism is designed to determine the possibility of a sub-token that *must not be copied to follow* the current sub-token in the currently generated method name. The non-copy mechanism complements to the copy one in the way that it *pushes down the unlikely candidates* (with the sub-tokens not following a certain token) in the resulting ranked list. Thus, the likely candidates are pushed up in the list, improving suggestion accuracy.

We conducted several experiments to evaluate DEEPNAME in method name consistency checking and in method name recommending on two large datasets used in prior works with +2M and +14M methods [28]. For inconsistency checking, DEEPNAME outperformed the state-of-the-art approaches in Liu *et al.* [20] and MNire [28] by relatively 13.3% and 2.1% in recall, 34.9% and 19.6% in precision, and 25.4% and 11.9% in F-score. For method name suggestion, DEEPNAME improves relatively over the state-of-the-art approaches in both recall (8.8%–46.1%) and precision (1.8%–30.5%). There are 44.3% of the cases suggested by DEEPNAME that exactly match with the correct method names in the oracle, and 4.7% of those cases (i.e., 2.1% of total cases) do not appear in training data.

```
1 private void declareGrouping(BoltDeclarer boltDeclarer,
       Node parent, String streamId, GroupingInfo
       grouping) {
2   // the old inconsistent method name is declareStream
3   if (grouping == null) {
4     boltDeclarer.shuffleGrouping(parent.getComponentId(),
          streamId);
5   } else {
6     grouping.declareGrouping(boltDeclarer, parent.get
          ComponentId(), streamId, grouping.getFields());
7   }
8 }
```

Fig. 1: An Example of Inconsistent Method Name

This shows that DEEPNAME can learn to suggest the method names, rather than retrieving what have been stored. In total, there are 11.9% of the cases in which the names are not previously seen in the training data. The precision and recall of this set of generated names are 57.6% and 55.1% respectively. To assess usefulness, we made 50 pull requests (PRs) on the suggesting new names for the inconsistent methods as detected by DEEPNAME. Among them, 12 PRs were actually merged into the main branch, and 18 were approved for later merging. In total, there are 60% of the cases that team members agreed that our suggested names are more meaningful than the current names. This paper makes the following contributions:

A. Representation and Tool: A novel approach that uses both internal and interaction contexts for method name consistency checking and suggestion.

B. Novel technique: In DEEPNAME, we modify an RNN-based encoder-decoder to integrate a newly developed mechanism, called *Non-copy* mechanism, to help our model pushes correct candidates to the top, improving top-ranked accuracy.

C. Empirical Results: Our empirical evaluation shows that 1) DEEPNAME is useful and more accurate than the state-of-the-art approaches in real-world projects and in a live study; 2) all four contexts complement to one another and contribute much to high accuracy. Our replication package is in [1].

II. MOTIVATING EXAMPLES

A. Examples

Figure 1 shows an example in Apache Storm project having the method declareGrouping with an inconsistent name. It is used to declare a group information for a stream. In an earlier version, the method was given the name declareStream, which was deemed to be confusing and inaccurately reflecting the functionality of this method. Therefore, in a later version, a developer performed refactorings to rename the method into declareGrouping and at the same time performed code partition.

This example shows a common case in which during the course of software development, the name of a method has become confusing and inconsistent with its functionality. Thus, an automated tool to detect inconsistent method names is helpful for developers to avoid confusing and mistakes.

When the method name is identified as inconsistent, it is also useful to have a tool to recommend a new name for the method. There are several factors that a tool can leverage to

```
1  private Dimension calculateFlowLayout(boolean
       bDoChilds){
2    ...
3    if (getParent()!=null && getParent()... JViewport) {
4       JViewport viewport = (JViewport) getParent();
5       maxWidth = viewport.getExtentSize().width;
6    } else if (getParent() != null){
7       maxWidth = getParent().getWidth();
8    } else {
9       maxWidth = getWidth();
10   }
11   ...
12   Dimension d = m.getPreferredSize();
13   ...
14 }
15
16 public Dimension XXXXXXXXXXXXXXXX() {
17   // The consistent method name is getPreferredSize
18   return calculateFlowLayout(false);
19 }
```

Fig. 2: An Example of Method Name Suggestion

suggest a new name for the method. First, a tool can rely on the body (i.e., the content) of the method to suggest its name. Second, the types and names of the arguments and return type of the method could also be used to predict the method's name. The first and second factors are referred to as the **internal** content and the **interface** of the method under study. These two factors represent the only two key sources of information that the state-of-the-art approaches have been using for method name checking/suggestion. Liu *et al.* [20] use clone detection on the methods' bodies to search for similar methods to suggest similar names. Alon *et al.* [7] also rely on the method's content, however, explore code structures by using embeddings built from the paths over the abstract syntax tree (AST) of the method under study. Allamanis *et al.* [3], [4] and Nguyen *et al.* [28] also make use of the method's body, method interface, and class name, however, break down the names of program entities into sub-tokens, and then use them to suggest the method name via neural network models [4], [28] or a clustering algorithm in the vector space [3].

Despite their successes, the state-of-the-art approaches do not work for the methods with little contents in their bodies. The method at line 16 in Fig. 2 in Tina POS project is named getPreferredSize. The method contains a single call to calculateFlowLayout. Assume that one wants to use an existing name suggestion tool for the body of this method. However, the existing approaches relying on the method's body or interface do not work because 1) none of the tokens of the correct name (getPreferredSize) appears there; 2) the code structure in the body does not help in predicting the method's name. Our tool suggests the correct name getPreferredSize, while MNire [28] uses the body to suggest the name getFlow. *Thus, simply using the method's body and interface is not sufficient.*

B. Key Ideas

1) **Show Me Your Friends, I'll Tell You Who You Are***:* In addition to the method's body and interface, we characterize a method by the surrounding methods that interact with the method under study. In this problem, the surrounding and neighboring methods of a method m under study could include the methods that are called within the body of m (let us call them *callees*), the methods that are calling m (*callers*), the methods within the same class with m (*siblings*), and the program elements declared in the enclosing class of m. For method name consistency checking, in addition to the method's body and interface, we could use all of those neighboring methods. For name suggestion, we could use callees and siblings since the callers of m might not be written yet at the time that the current method m is being edited.

In this example, while the content of m is short, the callee context, i.e., the body of the method calculateFlowLayout, contains sufficient information for name suggestion. In Fig. 2, examining the body of the callee method calculateFlowLayout, we can see that the sub-tokens of the consistent method name getPreferredSize appear in the names of the program entities in the callee. First, the sub-token get appears at lines 3-7, 9, and 12. Second, the sub-token Preferred appears at line 12. Third, the sub-token Size appears at line 5 and line 12. For consistency checking, callers and sibling methods can be used since they might be available. In general, the contexts complement to one another and to the internal content of the method, contributing to name suggestion. With the nature of source code, the case of little internal content and little contexts of a method is rare.

2) **Representation Learning from Multiple Contexts:** Our model learns the representation to integrate names of variables, fields, method calls from multiple contexts. In addition to the method's body and interface (we call it *internal context*), we also consider the *interaction context*, which includes all the methods interacting with the current method m, i.e., caller methods (if available) and callee methods. In Fig. 1, the two sub-tokens declare and Grouping in the consistent method name declareGrouping can be found at line 4 and line 6. We also use the sibling methods in the same class (*sibling context*) because they provide the tasks with the same theme.

Different contexts might have the sequences of sub-tokens with different lengths and nature. For example, in some cases, those sequences for callers/callees might positively contribute or negatively impact in deriving the method name. To help our model learn the importance of different contexts in different situations, we use a learning scheme for the weights of the contexts. In Fig. 2, we collect the internal context including the body at line 18 and the method return type Dimension at line 16 as one type of context. We collect the name of the method calculateFlowLayout. Also, we collect the body and interface of the method calculateFlowLayout as the interaction context.

3) **Sub-token Copy and Non-copy Mechanisms:** It is reported that the high percentage of sub-tokens of the method names can be found in the set of the sub-tokens of the program entities in the context. Thus, we leverage this by using a copy mechanism for Recurrent Neural Network (RNN). This emphasizes on the likelihood of directly copying a sub-token from the input into the output position following a sub-token.

We also design a *Non-copy* mechanism that complements to the copy one. As copying at a sub-token position, the non-

copy mechanism estimates the likelihood score that *a certain sub-token s must not be copied to follow the current sub-token c*. A higher score implies that the occurrence probability of the sequence consisting of c following s is lower. Thus, it helps our model push lower in the resulting ranked list the incorrect candidates with those sub-tokens that must not follow certain ones. The correct candidates thus are pushed higher in the list, improving top-ranked accuracy (Table 5). For example, as deriving the sub-token at the second position in the method name, *Non-copy* pushes the candidates declareSchool or declareStudent lower in the list since School and Student have never been seen to follow declare in the training set. Thus, the correct option declareGrouping will be ranked higher.

Importantly, instead of directly using RNN encoder-decoder, we modify its operations to integrate our newly developed *Non-copy* mechanism with the *Copy* mechanism to help better predict the method name. The *Copy* and *Non-copy* mechanisms complement each other. Assume that the currently predicted name has two sub-tokens: A B. *Copy* mechanism suggests $C_1, C_2, ..., C_n$ (from the input) to likely follow A B. That is, *Copy* mechanism suggests C_1 is likely to be copied, and C_n is less. However, *Non-copy* mechanism can suggest that C_2 (from the input) must not follow B because in training, it has never seen B C_2. Thus, with only *Copy* mechanism, C_2 is ranked 2nd, however, with both, we lower the ranking of C_2.

III. Concepts and Approach Overview

Definition 1 (Caller and Callee Methods). *A caller method (caller for short) is a method calling the current method under study for consistency checking or name suggestion. A callee method (callee for short) is a method called within the body of the current method.*

Definition 2 (Contexts). *We consider four types of context:*

1. Internal Context: *The internal context for the current method contains the content in the* **body**, **types** *and* **names** *of the* **arguments** *in the* **interface**, *and the method's* **return type**.

2. Interaction Context: *The interaction context for the current method includes 1) the* **names** *of the* **callers** *of the current method, 2) the* **contents** *of the callers (i.e.,* **bodies**, **interfaces**, **return types**), *3) the names of the* **callees**, *and 4) the contents of the callees (bodies, interfaces, return types).*

3. Sibling Context: *The sibling context for the current method m includes 1) the* **names** *of the methods in the same class with m, and 2) the* **contents** *of sibling methods (including the* **bodies**, **interfaces**, *and* **return types**).

4. Enclosing Context: *The enclosing context for the current method includes 1) the name of the enclosing class of the current method and 2) the names of the program* **entities**, **method calls**, **field accesses**, **variables**, *and* **constants** *in the class.*

Figure 3 shows DeepName's overview. It is aimed to help in two usage scenarios. First, the project is in the development process and the source code of the current method under consideration and the source code of the callers, callees, siblings, and the enclosing class are available. DeepName can be used to determine the consistency of its name, and then it is used

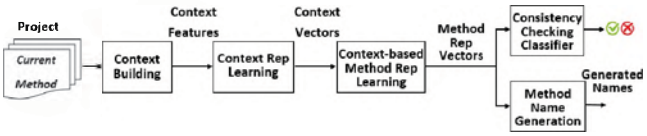

Fig. 3: Overview of DeepName's Architecture

to suggest an alternative name if the current name is deemed inconsistent. In the second scenario, the source code of the current method is written, however, its caller methods are not available since its name is not there yet. DeepName can be used to suggest a proper name for that method in this scenario.

There are four main steps in DeepName:

1. Context Building. We collect internal context, interaction context, sibling context, and enclosing context to build context features. Specifically, for each type of context, we extract the names of the program entities appearing in the context. We use the CamelCase and Hungarian convention to break each name into a sequence of sub-tokens. For example, calculateFlowLayout is broken into calculate, Flow, Layout. We use the sequence of all the sub-tokens of the program entities' names in the same order in the source code as the feature for that context. The rationale of using the sequence of sub-tokens as feature (instead of PDG/AST) is the naturalness of names [28]: developers do not give random names; they use names of program entities or method calls/fields relevant to the task of the current method.

2. Context Representation Learning. After extracting as features the sequences of sub-tokens for the contexts, we need to convert those sequences into vector representations for the models in the later steps. We use a word embedding model to convert the sequences of sub-tokens into the vectors. We put the vectors for all the sub-tokens in the order of the appearances of those sub-tokens in the source code. As the result, for each context, we have a sequence of representation vectors corresponding to the sub-token sequence of that context.

3. Context-based Method Representation Learning. DeepName relies on all four contexts (internal, interaction, sibling, enclosing). Thus, we need to produce a representation for method m with the encoding of all contexts. From the previous step, for a context, we have a sequence of representation vectors. At this step, we use an encoder to learn the encoding of the features for a context. We then use a decoder to combine the encoded information for all the contexts into a representation for m, which has the integration of all contexts.

4. Consistency Checking and Method Name Suggestion. At the last step, DeepName has two separate models for two tasks. For consistency checking, it takes as the input the representation of the method m obtained from the previous step with the existing method name to be checked, and feeds to a two-channel CNN-based model [39] acting as a classifier to classify the method name to be consistent or not. For method name suggestion, each vector in the sequence of method representation vectors is a sub-token representation vector. We use our vocabulary to roll back all the vectors into the sub-tokens and put them together to generate the method name.

IV. Context Representation Learning

A. Context Building

Our first step is to build the contexts from source code. For the internal context, sibling context, and enclosing context, we directly extract the names from the program entities, the return type, and the types in the interface. The names are broken into sub-tokens, which are collected into the sequence in the same appearance order in source code. The sequences of those sub-tokens represent the contexts. For example, for the method in Fig. 2 (line 16), the internal context is modeled by the sequence of sub-tokens: calculate flow layout dimension. The internal context also includes the sub-tokens in the return type, the types and names of the parameters of the method. The trivial (sub)tokens with a single character are removed.

For the interaction context, we build a call graph using Soot [33]. We then identify the callers and callees for the current method under study. The sequences of sub-tokens are built in the same manner as in the previous contexts to form the feature for the interaction context. For consistency checking, both callers and callees are considered. However, for method name suggestion, we consider only the callee methods if the caller methods do not exist yet. In Fig. 2, the callee part of the interaction context includes the sub-tokens built from parsing the method calculateFlowLayout:

get parent ... view port ... max width dimension get prefer size boolean dimension

Other callees are processed in the same manner. All the sequences of sub-tokens for all contexts are used as input in the next step. We denote those sequences as *context features*.

B. Context Representation Learning

To convert the sequences of sub-tokens into vectors, we consider all the sequences of sub-tokens for all the contexts as the sentences of words. We use *GloVe* [29], a word embedding technique, to produce the vectors for the collected sub-tokens. We use *GloVe*, instead of *Word2Vec*, due to its capability of learning to represent the words from the aggregated global word-word co-occurrence statistics, which captures the relationships between neighboring sub-tokens.

To build the vector representation for each context, we replace the sub-tokens in a sequence for the context feature with the corresponding *GloVe* vectors for the sub-tokens. We maintain the same order as the appearance order in the source code. For example, for a context feature sequence F_i, we have a sequence of vectors V_{F_i} = [v_1, v_2, ..., v_n], in which v_i is a *GloVe* vector for a sub-token in the sequence. Because the sequences of vectors might have different lengths, we perform zero padding by filling the zero vectors for the sequences whose lengths are less than the maximum length. This makes the sequences of vectors have the same length.

V. Context-Based Method Representation

From the previous step, a context is represented as a sequence of vectors in which each vector represents a sub-token. The goal of this step (Fig. 4) is to produce a representation for the given method that integrates all of its contexts.

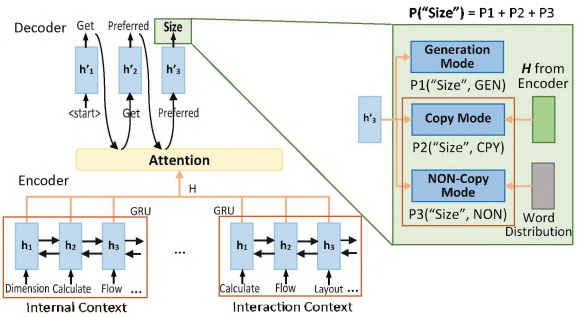

Fig. 4: Context-based Method Representation Learning

A. Context Feature Encoding

We first use a RNN-based *seq2seq* encoder to encode the sets of vectors for all contexts from the previous step. For each context, we encode it with a Gate Recurrent Unit (GRU). We use multiple GRUs because different contexts might have different structures and types of information. Multiple GRUs also help reduce the cross influence between different contexts.

The input for each GRU$_i$ is the sequence V_{F_i} of n vectors representing a context. Each vector represents a sub-token in the names of the program entities in the context. For example, in Fig. 4, one GRU is used for the interaction context including the sub-tokens in the name, body, and interface of the callee method: Calculate Flow Layout .... Another GRU is used for the internal context including the body/interface of the current method: Dimension Calculate Flow ... For each time step t, we input one vector V_t in these n vectors and we get one hidden state vector h_t as the output for this time step. By collecting all outputs for each time step, we have a list of hidden state vectors $H_i = [h_1, ..., h_n]$, which is the output of GRU_i.

Mathematically, the input sequence $V_{F_i} = [v_1, v_2, ..., v_n]$ is learned and converted into a hidden vector H_i by the encoder. The decoder is used to transfer the representation hidden vector H_i back to the target sequence $Y = [y_1, y_2, ..., y_m]$.

$$h_t = f(v_t, h_{t-1}) \tag{1}$$

$$h'_x t = g(y_{t-1}, h'_{t-1}, H_i) \tag{2}$$

$$p(y_t|y_1, ..., y_t - 1, V_{F_i}) = s(y_{t-1}, h_t, H_i) \tag{3}$$

Formula 1 is for encoder RNN. h_t is the hidden state in time step t; f represents the RNN dynamic function. Formula 2 is for the decoder. h'_t is the hidden state for the decoder at time step t; g represents the RNN dynamic function. Formula 3 is for prediction: s is the possibility calculation function.

By putting together all the outputs H_is of all GRUs, we obtain all the hidden states vectors $H = [H_1, H_2, ..., H_i]$. This is the output of the encoder and used in the attention layer.

Because not all the sub-tokens in a context are equally important, we aim to put emphasis on certain sub-tokens. Such emphasis is learned via an **attention mechanism**. Technically, the attention layer uses a changing context C_t instead of one

hidden vector H_i. C_t is calculated as the weighted sum of the encoder's hidden states:

$$C_t = \sum_{i=1}^{n} \alpha_{t,i} h_i \qquad \alpha_{t,i} = \frac{e^{r(h'_{t-1}, h_i)}}{\sum_{i' \neq i}^{n} e^{r(h'_{t-1}, h_{i'})}} \tag{4}$$

where r is the function used to represent the strength for attention, approximated by a multi-layer neural network.

B. *Context Feature Decoding*

This is our new component (Fig. 4) in which we modify the operation of the GRUs at the output layer of the decoder to integrate our newly developed *Non-copy* mechanism. It operates in connection with the attention layer.

First, at each time step t for a GRU, the previous hidden state h'_{t-1} is used as the input for the attention layer and the output of the attention layer will be used as the input of the GRU at the time step t. In Fig. 4, the vector for the sub-token Preferred obtained from the output of the attention layer at the time step 2 is used as the input of the GRU at time step 3. This emphasizes on the important sub-tokens while the decoding is performed at each time step.

Next, at the output layer of the decoder, we also integrate the operation of two mechanisms: **CopyNet** [11] and **Non-copy**.

1) **CopyNet***: CopyNet* [11] is the copy mechanism with the RNN *seq2seq* model and attention mechanism. It calculates the possibilities of copying the input sub-tokens to the output. It has two modes: generation-mode (denoted by GEN) and copy-mode (CPY) for prediction. The attention mechanism with RNN uses one function. The state update for *CopyNet* considers not only the word embedding, but also the corresponding location-specific hidden state in the set of RNN encoder's hidden states $H = [h_1, h_2, ..., h_t]$.

$$p(y_t|h'_t, y_{t-1}, C_t, H) = p(y_t, GEN|h'_t, y_{t-1}, C_t, H) + p(y_t, CPY|h'_t, y_{t-1}, C_t, H) \tag{5}$$

$$p(y_t, GEN|.) = \begin{cases} \frac{1}{Z} e^{u_{GEN}(y_t)} & y_t \in dic_{all} \\ 0 & y_t \in \overline{dic_{all}} \cap dic_{in} \\ \frac{1}{Z} e^{u_{GEN}(UNK)} & y_t \notin dic_{all} \cup dic_{in} \end{cases} \tag{6}$$

$$p(y_t, CPY|.) = \begin{cases} \frac{1}{Z} \sum_{j:v_j = y_t} e^{u_{CPY}(v_j)} & y_t \in dic_{in} \\ 0 & otherwise \end{cases} \tag{7}$$

u_{GEN} and u_{CPY} are the score functions for the generation-mode and copy-mode; Z is the normalized term used by two modes; and dic_{all} and dic_{in} are the overall dictionary and the dictionary for the input only.

2) **Non-copy mechanism:** We aim to determine the possibility of a sub-token that must *not* follow the current sub-token. For instance, if the sub-token get at the (m-1) position is a known one, and the sub-token Preferred follows it in the training dataset, we calculate the possibility that Preferred will *not* follow the sub-token get. Such calculation is based on our statistical analysis on the word distribution on the vocabulary.

Mathematically, for the method name y_{m-1} at the position (m-1), we calculate the possibility that a certain sub-token will not be the next one at the position m. That possibility score, denoted by $p(y_m, NON|y_{m-1})$, is computed as

$$p(y_m, NON|y_{m-1}) = \begin{cases} 1 - \frac{count_{(y_m|y_{m-1})}}{count_{y_{m-1}}} & y_m, y_{m-1} \in dic_{all} \\ 0 & y_{m-1} \notin dic_{all} \\ 1 & others \end{cases} \tag{8}$$

Where $count_{(y_m|y_{m-1})}$ is the occurrence count for the sub-token m that follows the sub-token at (m-1) in the training dataset, and $count_{y_{m-1}}$ is the occurrence count for the sub-token at (m-1). This formula is to calculate the word distribution possibility, which is between [0,1]. The larger value means that this sub-token has higher possibility of *not* following the previous sub-token at (m-1). Moreover, because we have multiple context features as input, which can pass to the copy mode, we add all the possibilities together as the total copy-mode possibilities with different weights. With this, the copy-mode Formula 7 has now become:

$$p(y_t, NEW|.) = \sum_{i=1}^{I} W_i p_i(y_t, CPY|.) + W_{NON} p(y_t, NON|y_{t-1}) \tag{9}$$

Where W_i is a trainable weight for different types of context features, W_{NON} is a trainable weight and always less than zero, and I is the total number of types of context features.

3) ***Method Representation Decoding****:* We have modified the decoder part (see our new component in Fig. 4) to integrate the copy and non-copy mechanisms. In the traditional GRU for an RNN decoder, the output layer is computed according to Formula 2. We still keep that computation as one of the three factors to determine the output of the decoder, which is shown as Generation Mode in Fig. 2. For example, for the sub-token Size after Preferred, that computation is as P(Size,GEN). In addition, we also integrate the *CopyNet* mechanism to determine the possibility of a sub-token based on the sub-token copying as in Formula 7 in Section V-B1: P(Size, CPY). For *Non-copy* mechanism, we integrate with the copy mechanism from Formula 8. Thus, the new formula for the combination of copy and non-copy mechanisms is Formula 9.

The possibility score of a sub-token as the output of the decoder at a time step t is the summation of the three factors. Thus, the possibility score of a sub-token, e.g., w=Size, is calculated as P(w) = P1(w,GEN) + P2(w,CPY) + P3(w,NON). We will pick the representation vector for the sub-token with the highest possibility score in the current time step t as the output of the decoder at that time step.

Finally, after having the prediction vector V_t for all time steps, we put them together to obtain a set of vectors in the original order as the set of vectors V_{cur} for the current method.

VI. Consistency Checking and Name Suggestion

A. *Consistency Checking*

To check whether the method name is (in)consistent, we use a two-channel CNN model [39] as a classifier on the set of vectors V_{cur}, which can be viewed as a matrix M_{cur}. To build the second channel, we apply the same word embedding step to represent the name of the current method as the set

of vectors V_{exist}. We consider that set as the matrix M_{exist} and combine with the matrix M_{cur} to form the two-channel representation matrix $M_{classification} = [M_{cur}, M_{exist}]$, which is fed to the two-channel CNN model. The output is produced by the *softmax* function. The value is between [0-1] in which 1 represents consistency and 0 represents inconsistency.

B. Method Name Suggestion

We consider the sequence V_{cur} produced by the previous step is the vector representations for the method name under study. Specifically, we consider each vector V_k as the representation for a sub-token in the suggested name of the method. From the dictionary dic_{all} for all the vocabulary in the corpus, we find the closest vector V_{s_k} to the vector V_k and use the sub-token s_k corresponding to V_{s_k} as the suggested sub-token for V_k. Finally, we obtain the sequence of the sub-tokens for all V_ks. The resulting sequence is considered as the suggested name for the method under study. The order of the sub-tokens is the same as the order of the vectors V_ks in the sequence of the representation vectors for the method V_{cur}.

VII. Empirical Evaluation

A. Research Questions

RQ1. *Method Name Consistency Checking Comparative Study.* How well does DeepName perform in comparison with the state-of-the-art method consistency checking approaches?
RQ2. *Method Name Suggestion Comparative Study.* How does DeepName perform in comparison with the state-of-the-art method name suggestion approaches?
RQ3. *Impact Analysis of Different Contexts and Weighting Scheme.* How do distinct types of contexts and their different weights affect the overall performance of DeepName?
RQ4. *Impact Analysis of Copy and Non-copy Mechanisms.* How do copy and non-copy mechanisms affect accuracy?
RQ5. *Method Name Suggestion Accuracy Analysis.* How does DeepName perform on the un-seen method names and the methods in various sizes?
RQ6. *Live Study.* How does DeepName perform on the currently active real-world projects?

B. Experimental Methodology

*1) **Datasets:*** **Corpus for Consistency Checking.** For comparison, we used the same dataset from Liu *et al.* [20], which was also used in another baseline approach MNire [28]. The training dataset (Table I) from that corpus was collected from the highly-rated, open-source projects from four communities, namely *Apache*, *Spring*, *Hibernate*, and *Google*. It contains the latest versions of 430 Java projects with +100 commits. In total, it has 2,119,573 methods, which were considered as consistent names because the methods whose names are stable for a long time were selected. For the testing dataset, Liu *et al.* [20] mined the methods whose names were modified by developers for the reasons of misleading names. Finally, in those projects, there are 1,402 methods with inconsistent names. They randomly chose another 1,402 methods with consistent names to form a test dataset with 2,804 methods.

TABLE I: Corpus for Method Name Consistency Checking

	Testing data	Training data
#Methods	2,804	2,119,573
#Files	–	251,362
#Projects	–	430
#Unique method names	–	540,547

TABLE II: Corpus for Method Name Suggestion

	Testing data	Training data	Total
#Project	1,022	9,200	10,222
#File	51,631	1,756,282	1,807,913
#Methods	466,800	13,992,028	14,458,828

Corpus for Method Name Suggestion. For comparison, we used the same dataset as in code2vec [7] and MNire [28], with 10,222 top-ranked Java projects from GitHub. It has 14,458,828 methods and 1,807,913 unique files. We split the corpus based on the number of projects, instead of files. The project-based setting reflects better the real-world usage of DeepName where it is trained on the set of existing projects and used to check for a new project. The overloading/overriding methods and generated method names were removed.

*2) **Empirical Procedure:*** Let us present our procedure.
RQ1. Method Name Consistency Checking. We chose the following baselines: 1) *Liu* et al. [20], an IR approach to search for similar methods to suggest similar names, 2) *MNire* [28], an ML approach using seq2seq encoder-decoder on the sub-token sequences in the method's body and interface. We trained each model under study with the same training dataset and tested it with the same testing dataset.

For hyper-parameter tuning for a model, we used AutoML in NNI [22] to automatically tune the parameters. We selected the parameter set that helps a model obtain the highest F-score and accuracy. For name consistency checking, 2,804 methods are for testing; 90% (1,907,716 methods) of the training data are for training; and the remaining 10% are for tuning.

RQ2. Method Name Suggestion. We compared with the following baseline approaches: 1) **MNire** [28], 2) **code2vec** [7] 3) **code2seq** [5], and 4) **path-based representation** [6]. For comparison, using the same procedure as MNire [28], we randomly split the data by 80% for training, 10% for parameter tuning, and 10% for testing.

RQ3-RQ4. Impact Analysis on various Components. We varied our model (e.g., adding each context), and measured accuracy. We used the parameter settings as in RQ1-RQ2.

RQ5. Accuracy Analysis on Method Name Suggestion. We measure the accuracy on un-seen method names and on the methods with different lengths in the same setting as RQ2.

RQ6. Live Study. We use our tool to check the method names in the active GitHub projects, make pull requests to rename inconsistent ones, and evaluate the acceptance responses.

*3) **Evaluation Metrics:*** For method name consistency checking, we compared the predicted results against the ground truth on consistent and inconsistent method names provided as part of the name consistency checking corpus [20]

TABLE III: RQ1. Method Consistency Checking Comparison (C: Consistent Methods; IC: Inconsistent Methods)

		Liu *et al.* [20]	MNire [28]	DEEPNAME
C	Precision	46.5%	54.1%	64.8%
	Recall	68.3%	80.6%	86.4%
	F-score	55.3%	64.7%	74.1%
IC	Precision	53.6%	60.5%	72.3%
	Recall	81.3%	90.2%	92.1%
	F-score	64.6%	72.4%	81.0%
Accuracy		56.8%	65.7%	75.8%

(Table I). For name suggestion, we compared the predicted names by a model against the good method names in the name suggestion corpus, which was part of code2vec [7] (Table II).

For consistency checking, we used the same evaluation metrics as in Liu *et al.* [20] and MNire [28] including Precision, Recall, and F-score, for both inconsistency (IC) and consistency (C) classes. For IC class, Precision = $\frac{|TP|}{|TP|+|FP|}$, and Recall = $\frac{|TP|}{|TP|+|FN|}$. For C class, Precision = $\frac{|TN|}{|TN|+|FN|}$, and Recall = $\frac{|TN|}{|TN|+|FP|}$, in which TP is true positive (IC is classified as IC), FN is false negative (IC is classified as C), TN is true negative (C is classified as C), and FP is false positive (C is classified as IC). For both IC and C, F-score is defined as $\frac{2\times Precision\times Recall}{Precision+Recall}$. For the overall in both IC and C, Accuracy is defined as $\frac{|TP|+|TN|}{|TP|+|FP|+|TN|+|FN|} = |TP| + |TN|$.

For method name suggestion, we used the same metrics as in code2vec [7] and MNire [28]: Precision, Recall, and F-score over case-insensitive sub-tokens. That is, for the pair of an expected method name e and its recommended name r, precision and recall are computed as: Precision(e, r)= $\frac{|subtoken(r)\cap subtoken(e)|}{|subtoken(r)|}$, and Recall(e, r)= $\frac{|subtoken(r)\cap subtoken(e)|}{|subtoken(e)|}$; $subtoken(n)$ returns the sub-tokens in the name n. Precision, Recall, and F-score of *the set* of the suggested names are defined as the average values in all cases. In all experiments, we also counted the exact-matched names (ExMatch) and the case-sensitive names.

C. Experimental Results

1) **RQ1. Method Name Consistency Checking**: As seen in Table III, for **inconsistent method name detection (IC)**, DEEPNAME has a relative improvement of (35.0%, 13.3%, 25.5%) and (19.6%, 2.1%, 11.9%) on (Precision, Recall, and F-score) in comparison with Liu *et al.* [20] and MNire [28], respectively. We found that for Liu *et al.* [20], in many cases, the inconsistent methods might not be classified as inconsistent (lower Recall), and the predicted inconsistent methods might be incorrect (lower Precision). The main reason is that the principle of "methods with similar bodies have similar names and vice versa" does not hold in many cases. For MNire, several inconsistent methods are not classified as inconsistent (lower recall) since the bodies use similar sub-tokens, but the methods do not have the same tasks. For those cases, DEEPNAME uses the caller and callee methods to complement for the internal context in the characterization of a method, and is able to detect the inconsistencies since it can detect methods with the same usages but with different names.

TABLE IV: RQ2. Method Name Suggestion Comparison

	code2vec [7]	Path-Rep [6]	code2seq [5]	MNire [28]	DEEPNAME
ExMatch	21.7%	23.3%	32.4%	38.9%	**44.3%**
Precision	60.2%	56.4%	72.3%	67.4%	73.6%
Recall	52.4%	49.2%	66.1%	63.1%	71.9%
F-score	56.0%	52.6%	69.1%	65.2%	72.7%

For **consistent method name detection (C)**, DEEPNAME has a relative improvement of (39.5%, 26.5%, 33.9%) and (19.8%, 7.1%, 14.4%) on Precision, Recall, and F-score in comparison with Liu *et al.* [20] and MNire [28], respectively.

Regarding **accuracy** as considering **both consistent and inconsistent name detection**, DEEPNAME relatively improves *33.6%* and *15.4% compared to Liu* et al. *[20] and MNire [28]*, respectively. We found several consistent methods with similar bodies, however with different names. For example, the methods on InputStream and OutputStream, but have the same body of return stream;. Both Liu *et al.* [20] and MNire [28] relies on the body, thus, cannot work in those cases, while DEEPNAME distinguishes them via callers/callees and siblings. Moreover, there are methods with the same bodies but names are different due to different enclosing classes intended for different purposes, e.g., process.start(), process.stop(). The baselines detected them as inconsistent. DEEPNAME uses the callers/callees to recognize its usage, thus, correctly detecting it as consistent.

2) **RQ2. Method Name Suggestion**: As seen in Table IV, **44.3%** of the cases suggested by DEEPNAME at top-1 positions are exactly matched with the correct method names given by developers. It has relative improvements from **13.9%–104.1%** compared with the baselines. It also achieves higher F-score than all the baselines. Specifically, it has relative improvements of 1.8%–30.5% in Precision, 8.8%–46.1% in Recall, and 5.2%–38.2% in F-score over the baselines.

With regard to ExMatch, code2vec and path-based Rep have lower values than DEEPNAME as they mainly encode path-based contexts with tokens, which have shown as less repetitive than the sub-tokens [28]. code2seq encodes the path-based contexts as well as the sub-tokens. However, it failed to capture the order of sub-tokens for the exact name recommendation. code2seq often recommends the relevant sub-tokens, but not in the right order. DEEPNAME improves code2seq by 37%. Also MNire does not consider the callers/callees and siblings, thus it cannot identify more sub-tokens than DEEPNAME.

With regard to Recall, we found that code2vec and path-based Rep have lower recall than DEEPNAME because the baselines mainly encode path-based contexts within one method. code2seq requires two methods with similar sub-tokens and/or path contexts. MNire requires two methods with similar sequences of sub-tokens to have similar names. DEEPNAME uses the bodies/interfaces as well as the interaction, sibling, and enclosing class contexts, thus, is more flexible.

With regard to Precision, two methods can be realized in the same structure, but are named differently since they are in different classes and are used differently. Because two methods have the same/similar AST path contexts, code2vec and Path-based Rep suggest the same name, thus, they have lower preci-

TABLE V: RQ3. Context Analysis on Consistency Checking

		Internal (A)	A+Enclosing (B)	B+Siblings (C)	C+Interaction (DEEPNAME)
C	Precision	53.1%	54.4%	56.5%	64.8%
	Recall	79.3%	80.9%	81.9%	86.4%
	F-score	63.6%	65.1%	66.9%	74.1%
IC	Precision	59.2%	61.1%	63.4%	72.3%
	Recall	88.4%	89.1%	89.3%	92.1%
	F-score	70.9%	72.5%	74.2%	81.0%
Accuracy		64.2%	65.3%	67.3%	75.8%

TABLE VI: RQ3. Context Analysis on Name Suggestion

	Internal (A)	A+Enclosing (B)	B+Siblings (C)	C+Interaction (DEEPNAME)
ExMatch	38.3%	38.8%	39.7%	44.3%
Precision	65.7%	66.3%	68.5%	73.6%
Recall	62.4%	63.9%	65.7%	71.9%
F-score	64.0%	65.1%	67.1%	72.7%

TABLE VII: RQ4. *Copy/Non-Copy* in Consistency Checking

		Seq2seq	Seq2seq +Copy	Seq2seq+Copy+Non-copy (=DEEPNAME)
C	Precision	64.7%	69.8%	72.3%
	Recall	89.5%	91.2%	92.1%
	F-score	75.1%	79.1%	81.0%
IC	Precision	57.3%	59.6%	64.8%
	Recall	82.4%	83.1%	86.4%
	F-score	67.6%	72.1%	74.1%
Accuracy		68.8%	73.5%	75.8%

TABLE VIII: RQ4. *Copy/Non-Copy* in Name Suggestion

	Seq2seq	Seq2seq+Copy	Seq2seq+Copy+Non-copy (=DEEPNAME)
ExMatch	39.4%	42.6%	44.3%
Precision	69.3%	72.2%	73.6%
Recall	68.5%	70.4%	71.9%
F-score	68.9%	71.3%	72.7%

sions. MNire uses the enclosing class but does not consider the interaction and sibling contexts. Thus, in several such cases, MNire suggests the same name, while DEEPNAME suggests the correct name due to the callers/callees and siblings.

3) **RQ3. Impact Analysis of Different Contexts and Weights***:* **A. Context Analysis.** The base model in this experiment uses only internal context (the method body and interface). As seen in Table V, when adding the enclosing class of the method (A+enclosing), accuracy increases by 1.1% (1.7% relatively), as both F-scores for C and IC classes increase. Considering the sibling methods, accuracy additionally increases 2.0% (3.1% relatively) as comparing the columns (B) and (C). Finally, with all the contexts, accuracy additionally increases 8.5%, i.e., 12.6% relatively. Thus, all contexts contribute positively toward the overall accuracy.

With further analysis, we have observed the following. First, the *enclosing class* provides the context related to the general theme of the current method, e.g., InputStream versus OutputStream. While the method bodies are the same (return stream;), DEEPNAME is able to derive the correct names getInputStream and getOutputStream by leveraging the enclosing context. Second, the *sibling context* provides the names of the relevant methods to the current one. For example, in a class that provides mouse handling for a canvas, the sibling methods onMouseUp and onMouseDown give useful sub-tokens to suggest the method name onMouseOver. Finally, the interaction context helps suggest the names for the methods with little content in the bodies, e.g., in delegation methods. Unlike the existing approaches with only internal context, we leverage the interactions, siblings, and enclosing contexts of the method as well as internal context (body/interface) to achieve highest accuracy.

We also performed another experiment to leave one context out and compare the accuracy with DEEPNAME's accuracy in order to determine the impact of each context. Without the interaction context, accuracy and two F-scores reduce by 11.2%, 8.5% and 9.7%, respectively. Without the sibling context, the performance decreases by 8.8%, 6.5%, and 7.9% on accuracy and two F-scores, respectively. The enclosing context also positively contributes to high performance. In brief, the internal and interaction contexts contribute the most.

The contributions of contexts are also confirmed by the method name suggestion results (Table VI). When adding the enclosing context to the internal one, F-score increases by 1.1%. When further adding the sibling context, F-score additionally increases by 2.0%. Finally, adding the interaction context, F-score additionally contributes 5.6%.

B. Impact of Weight Learning for Different Contexts. *The nature and the length of the sequences of sub-tokens in each context might contribute differently*. To help our model learn the importance of each context, we use weight learning with W_i in Formula 9. We compared our model with the one having equal weights. The result shows that with weight learning, DEEPNAME improves 5.13% in accuracy for consistency checking and 2.2% in F-score for name suggestion. Thus, our weight learning for contexts positively contributes to accuracy.

4) **RQ4. Impact Analysis of *Copy* and *Non-Copy* Mechanisms:** In this experiment, we removed from our model both *Copy* and *Non-Copy* mechanisms and used it as a baseline. We then added each mechanism one-by-one to the baseline. As seen in Tables VII–VIII, *Copy* mechanism helps improve over the baseline model 6.8% relatively in accuracy for consistency checking, and 3.4% relatively in F-score in name suggestion.

The newly developed *Non-Copy* mechanism helps additionally improve 3.1% relatively over seq2seq+Copy in consistency checking accuracy, and helps improve 2.7% relatively in name suggestion. Thus, *Non-copy* complements to *Copy* mechanism.

5) **Illustration***:* Let us take an example in our experiment to illustrate the effectiveness of each component in DEEPNAME. Fig. 5 shows the top-ranked results for the name of the method given in Fig. 6 whose actual name is processFinallyStmt. We show the top-ranked resulting lists of the name for several variations of DEEPNAME in which we gradually added each component/context to the previous model. The leftmost column is the result of the model using only internal context

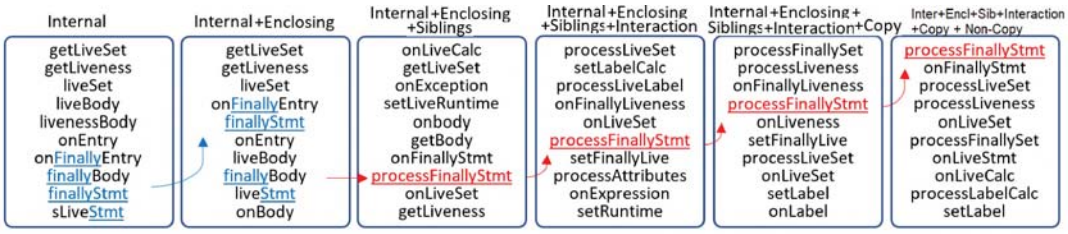

Fig. 5: Method Name Recommendation Results (Top-10 Ranked List) for Fig. 6

```
1 private static LiveCalc XXXXXXXXXX (FinallyStmt s,
       LiveSet onEntry) {
2    //Name: processFinallyStmt
3    return liveness(s.getBody(), onEntry);
4 }
```

Fig. 6: A Correctly Suggested Method Name

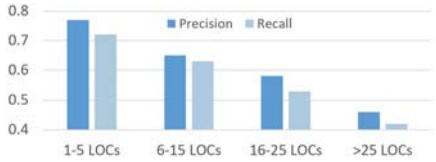

Fig. 7: Accuracy by different Methods' Sizes in Test Set

(body and interface). The next column is the result from a new variation with the addition of a new component. For example, the second column is the result from a model that considers both internal and enclosing contexts; the third one is from the model with internal, enclosing, and sibling contexts, etc.

Let us explain the resulting lists from all the variants and the impacts of contributing components. As seen, both the models Internal and Internal+Enclosing cannot suggest the correct name in the top-10 lists. The body and interface contain the sub-tokens Finally and Stmt, but do not contain the sub-token process. By adding the enclosing context can only help improve the ranking of the candidate method names that include sub-token Finally and Stmt. However, by adding the sibling context, Internal+Enclosing+Siblings is able to rank the correct method name processFinallyStmt at the 8th position. The reason is that the enclosing class in this case has several sibling methods with the names starting with process, e.g., processOperation, processLabeledStmtWrapper, etc. Thus, Internal+Enclosing+Siblings is able to learn the sub-token process from the sibling methods, and ranks the correct name higher.

By adding the interaction context, Internal+Enclosing+Siblings+Interaction can rank the correct name at the 6th position. The reason is that the body and interface of the callee method liveness contain the occurrences of process and stmts.

With the addition of the *Copy* mechanism, the new model Internal+Enclosing+Siblings+Interaction+Copy improves the ranking of the correct name to the 4th position. The reason is that *Copy* mechanism can emphasize on the copying of the popular and important sub-tokens, e.g., process, liveness. As seen, the names with the sub-tokens process and liveness are ranked at the top 5 positions in the column corresponding to this model.

Finally, DEEPNAME, with the addition of *Non-copy* mechanism, is able to rank the correct name at the top position. The reason is that *Non-copy* can learn that *the sub-token Set must not follow processFinally*. Thus, processFinallySet is pushed down, and the sub-token Stmt following processFinally to produce the correct name processFinallyStmt is pushed to the top.

TABLE IX: RQ7. Pull Requests of Real-world Projects

Accept	Agree	Disagree	No-reply	Total
12	18	11	9	50

6) **RQ5. Accuracy Analysis on Method Name Suggestion***:* We study the results on the suggested method names that *were not in the training data*. There are +173K (11.9%) out of +1,445K generated method name that were un-seen during the training. The Precision, Recall, and F-score for this set are 57.6%, 55.1%, and 56.3%, respectively. Importantly, in 17.4% of these generated cases (i.e., 2.1% total cases), the generated names exactly match the expected names in the oracle. These numbers show that DEEPNAME performs well for the un-seen method names and really learns to suggest natural names, rather than retrieving method names that have been stored in the training dataset.

Accuracy by the Sizes of Methods in Test Set. As seen in Fig. 7, DEEPNAME works well on the methods with the regular sizes of 1-25 LOCs. Even with the longer methods (+25 LOCs), Precision and Recall decrease gracefully at 46.3% and 41.9%, respectively. This shows that predicting the name becomes harder for longer methods. Even so, in +8K cases of 33K long methods with +25 LOCs, DEEPNAME produced the exact-match names with the correct ones.

7) **RQ6. Live Study***:* To evaluate the usefulness of our tool, we conducted a study on 100 randomly chosen, active Java projects in GitHub. We used DEEPNAME trained as in RQ1 to detect inconsistent method names, then submitted pull requests (PRs) of method renaming suggested by the tool and assessed PR acceptance rates. Overall, it identified 3K out of 133K methods as inconsistent. To avoid much work for developers, we randomly selected and made only 50 pull requests.

As seen in Table IX, among 50 PRs, 12 cases were approved and merged by the development teams. Additionally, 18 PRs have been validated and approved by the team members. For those cases, the teams acknowledged that the current method

names are misleading, and agreed with the suggested names as providing more meaningful names. However, at the time of writing, the PRs have not been merged into the main branch due to the additional requirements of reviewing or testing. In 11 cases, the developers disagreed with our suggested names. In some cases, the suggested names do not conform to coding conventions in the project. Some cases involve template code. In some cases, the names are of the methods that override the external libraries. There are still 9 cases that we did not get responses. In brief, in 30/50 cases, the developers confirmed that the names suggested by DEEPNAME are more meaningful than the current names. This shows that DEEPNAME is useful in real-world projects in both detecting inconsistent method names and suggesting new names.

8) Threats to Validity: Our data has only Java code. For code2vec [7], we used the same metrics for comparison (i.e., the accuracy for a set of method names are the average of those for all names). We did not have a statistical test in comparison since they did not provide individual resulting names. Running their tool requires a high-computational machine. We could not run Liu *et al.* [20]'s name suggestion on our dataset despite our efforts trying and contacting the authors without responses.

9) Limitations: Despite the above successes, DEEPNAME also has the aspects that need to be improved. As any other ML approaches, it has the out-of-vocabulary issue. That is, it cannot generate a sub-token that has never been seen in the training data. However, as shown in the empirical evaluation section, DEEPNAME is able to generate a new method name from the sub-tokens that it has encountered in the training dataset. Because DEEPNAME does not analyze the entire project, it does not perform well for the overriding methods, and the methods that override the APIs in the external libraries. A potential solution is to integrate our ML direction with program analysis to guide the process of the method name generation. Moreover, DEEPNAME does not work well for the method names with one single sub-token, or the methods with long bodies or the long callers/callees.

VIII. RELATED WORK

There are two categories of approaches for method name inconsistency detection and suggestion. The first one is Information Retrieval (IR). Liu *et al.* [20] relies on the principle that methods with similar bodies have similar names. In our experiment, we showed that such principle does not hold in many cases. Jiang *et al.* [18] searches for the methods having similar return type and parameters, as well as heuristics, to derive method names. The key advantage of these IR-based approaches is that they are light-weighted and do not require high computational power. Their key disadvantage is that because searching in the set of already-existed names, they cannot generate a new name that were not in the training data.

The second category is machine learning (ML). MNire [28] explores the sub-tokens appearing in the methods' bodies and interfaces. Allamanis *et al.* [4] use a neural network with attention and convolution to summarize code into descriptive summaries. Allamanis *et al* [3] project all the sub-tokens in the entities' names in the method bodies into the same vector space and cluster them to compose method name. MNire has been shown to outperform code2vec [7], which outperforms Allamanis *et al.* [4] and Allamanis *et al* [3]. Those ML-based approaches all rely only on the method's body and interface.

There are several approaches for code embeddings. Code2vec [7] abstracts source code by the paths over the AST to produce vectors. Code2seq [5] generates a word sequence from the structure of source code. Ke and Su [37]'s approach builds the embeddings to capture structures and semantics of a program. However, as shown in MNire [28], using code structure, AST, PDG is too strict in predicting method names.

There are several approaches to *predict the names or types of program entities* within the method bodies [30], [32], [34], [35]. While JSNeat [34] searches for names in a large corpus to recover variable names in minified code, JSNice [32] and JSNaughty [35] use CRF and machine translation. Naturalize [2] learns and enforces a consistent naming conventions.

Several approaches use ML to generate texts from code [14], [16], [17], [21], [36] and vice versa [12], [13], [23], [31], or code migration [24], [25], [26], [27]. Zheng *et al.* [38] uses AST structure for such statistical machine translation to produce comments. CODE-NN [17] uses LSTM on code sequence to model the conditional distribution of a summary to produce word by word. DeepCom [16] has a traversal on AST for flattening, and uses seq2seq to produce code summary. Wan *et al.* [36] use a deep reinforcement learning on AST and code sequence. There are several studies on name consistency and naming convention [8], [9], [15], [19].

IX. CONCLUSION

We introduce DEEPNAME, a context-based deep learning approach for inconsistency checking and method name suggestion. The following key ideas enable our approach (1) characterizing a method by the surrounding methods that are interaction or siblings method for the method we are studying; (2) learning the representation for the method with multiple contexts; (3) using sub-token copying and non-copying mechanisms to help better predict the name. We conducted several experiments to evaluate DEEPNAME.

Our results showed high accuracy and usefulness of DEEPNAME in real-world projects. For consistency checking, DEEPNAME improves the state-of-the-art approach by 2.1%, 19.6%, and 11.9% relatively in recall, precision, and F-score, respectively. For name suggestion, DEEPNAME improves relatively over the existing approaches in precision (1.8%–30.5%), recall (8.8%–46.1%), and F-score (5.2%–38.2%). In the assessment of DEEPNAME's usefulness in real-world projects, the team members agree that our suggested method names are more meaningful than the current names in 30/50 cases. For future work, we plan to integrate our ML direction with program analysis to improve both accuracy and efficiency.

ACKNOWLEDGMENT

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

[1] (2021) The github repository for this study. [Online]. Available: https://github.com/deepname2021icse/DeepName-2021-ICSE

[2] 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*, ser. FSE 2014. ACM Press, 2014, pp. 281–293.

[3] ——, "Suggesting accurate method and class names," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[4] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 2091–2100. [Online]. Available: http://proceedings.mlr.press/v48/allamanis16.html

[5] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 404–419. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192412

[7] ——, "Code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290353

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

[9] 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*, ser. MSR '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 203–206.

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

[11] J. Gu, Z. Lu, H. Li, and V. O. Li, "Incorporating copying mechanism in sequence-to-sequence learning," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1631–1640. [Online]. Available: https://www.aclweb.org/anthology/P16-1154

[12] 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*, ser. FSE 2016. ACM, 2016, pp. 631–642.

[13] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. ACM, 2015, pp. 416–432.

[14] T. Haije, "Automatic comment generation using a neural translation model," 2016.

[15] E. W. Host and B. M. Ostvold, "Debugging method names," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 294–317.

[16] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. ACM, 2018, pp. 200–210. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196334

[17] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: https://www.aclweb.org/anthology/P16-1195

[18] L. Jiang, H. Liu, and H. Jiang, "Machine learning based automated method name recommendation: How far are we," in *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE'19)*. IEEE CS, 2019.

[19] S. Kim and D. Kim, "Automatic identifier inconsistency detection using code dictionary," *Empirical Softw. Engg.*, vol. 21, no. 2, pp. 565–604, Apr. 2016.

[20] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. L. Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 41th International Conference on Software Engineering*, ser. ICSE '19. ACM, 2019, pp. 1–12.

[21] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, "Automatic generation of pull request descriptions," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, pp. 176–188. [Online]. Available: https://doi.org/10.1109/ASE.2019.00026

[22] Microsoft, "Neural network intelligence." https://github.com/microsoft/nni, last Accessed May 9th, 2020.

[23] A. T. Nguyen, P. C. Rigby, T. Nguyen, D. Palani, M. Karanfil, and T. N. Nguyen, "Statistical translation of english texts to api code templates," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 194–205.

[24] A. T. Nguyen, Z. Tu, and T. N. Nguyen, "Do contexts help in phrase-based, statistical source code migration?" in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 155–165.

[25] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining api usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 457–468. [Online]. Available: https://doi.org/10.1145/2642937.2643010

[26] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 651–654. [Online]. Available: https://doi.org/10.1145/2491411.2494584

[27] ——, "Divide-and-conquer approach for multi-phase statistical migration for source code," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, pp. 585–596. [Online]. Available: https://doi.org/10.1109/ASE.2015.74

[28] S. V. Nguyen, T. Le, and T. N. Nguyen, "Suggesting natural method names to check name consistencies," in *Proceedings of the 42th International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020, pp. 1–12.

[29] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[30] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen, "Statistical learning of API fully qualified names in code snippets of online forums," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. ACM, 2018, pp. 632–642.

[31] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing what i mean: Code search and idiomatic snippet synthesis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 357–367.

[32] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. ACM, 2015, pp. 111–124.

[33] Soot, "Soot introduction." https://sable.github.io/soot/, last Accessed July 11, 2019.

[34] H. Tran, N. Tran, S. Nguyen, H. Nguyen, and T. N. Nguyen, "Recovering variable names for minified code with usage contexts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE'19. IEEE Press, 2019, pp. 1165–1175.

[35] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. ACM, 2017, pp. 683–693.

[36] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," *CoRR*, vol. abs/1811.07234, 2018. [Online]. Available: http://arxiv.org/abs/1811.07234

[37] K. Wang and Z. Su, "Blended, precise semantic program embeddings," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 121Â¡V134. [Online]. Available: https://doi.org/10.1145/3385412.3385999

[38] M. L. Wenhao Zheng, Hongyu Zhou and J. Wu, "Codeattention: translating source code to comments by exploiting the code constructs," *Frontiers of Computer Science*, vol. 13, pp. 565–578, 2018.

[39] S. Zagoruyko and N. Komodakis, "Learning to compare image patches via convolutional neural networks," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 4353–4361.