Just-In-Time Method Name Updating With Heuristics and Neural Model

Zhenting Guo, Meng Yan*, Hongyan Li, Zhezhe Chen, and Weifeng Sun
School of Big Data and Software Engineering, Chongqing University, Chongqing, China
{cqugzt, mengy, hongyan.li, weifeng.sun}@cqu.edu.cn

2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS) | 979-8-3503-1958-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/QRS60937.2023.00074

Abstract—Ensuring the quality and conciseness of method names is pivotal for the readability and maintainability of source code. However, for developers, it often presents challenges, particularly during the course of code evolution. Throughout this process, developers sometimes may neglect to update the method name, resulting in inconsistency which could potentially mislead developers and introduce future bugs. In this paper, we propose the task of "Just-In-Time (JIT) Method Name Updating" which automatically performs method name updates to avoid inconsistent names and fix them before being introduced into code bases. Specifically, we propose an approach that combines heuristic rules and a neural model. The heuristic rule-based component mainly focuses on the single-token changes for our empirical findings that the proportion of single-token modifications is extensive, and often corresponds to code-indicative updates. The neural model-based component is a customized Seq2seq model considering code changes and the new method body's token type. To evaluate our approach, we conduct extensive experiments on the collected dataset with over 108K method name-body co-change samples from popular Java projects. The results show that our method outperforms the three baselines on all metrics. In particular, our approach achieves a significant improvement in Accuracy and improves method generation baseline by 23.5%.

Keywords–method name updating; seq2seq model; heuristic rule

1. Introduction

Method name plays a crucial role in code readability and software quality [15]–[17], [21], [47]. Developers rely heavily on identifiers for program comprehension, as they contain the semantic information related to programs [25], [42], [46]. Method name is a special type of identifier, which can be considered as concise documentation, abstractly summarizing the functionality of a program, e.g. get elements, set variables. There would be many function calls using corresponding method names during the implementation of a software project, which are always co-written by different developers. A good method name could facilitate communication between developers, while it is always difficult for developers to write consistent names in programs due to various reasons such as insufficient communication among development teams and lack of understanding of project development histories. That being the case, a misunderstanding or a misuse might occur in the presence of inconsistent method name, resulting in serious defects [4], [9], [14], [46]. Moreover, improper method names would decrease the quality of projects, causing difficulties for program maintenance and upgrades [11], [25], [28], [32], [44]. During the development of the project, developers may forget or ignore to update method name when making modifications to the method body, resulting in an inconsistent method name that requires fixing. As the example shown in Figure 1, which is extracted from the crawljax[1], function "click()" is changed to "lookFor()", but the developer does not perform the method name update and introduce it into the code bases. Such inconsistent method names may persist for an extended period, leading to potential confusion and time wastage among developers before being fixed. These inconsistent names can mislead developers, increasing the likelihood of introducing bugs [4], [9], [14] as well as additional time being spent on the implementation checking and code reviews [10], [11], [25], [28]. As such, it is crucial to address and rectify inconsistent method names right on time or, better yet, prevent their introduction. Hence, in this study, we first present the task of "Just-In-Time (JIT) Method Name Updating", where we aim to automatically generate candidate solutions with code changes to update inconsistent method names. Several studies have been

Inconsistent method name example

Before Editing:

```
public void clickDefaultElements() {
  crawlActions.click("a");
  crawlActions.click("button");
  crawlActions.click("input").withAttribute("type",
                                "submit");
  crawlActions.click("input").withAttribute("type",
                                "button");
}
```

After Editing:

```
public void clickDefaultElements() {
  crawlActions.lookFor("a");
  crawlActions.lookFor("button");
  crawlActions.lookFor("input").withAttribute("type",
                                "submit");
  crawlActions.lookFor("input").withAttribute("type",
                                "button");
}
```

After Fixing:

```
public void lookForDefaultElements() {
  crawlActions.lookFor("a");
  crawlActions.lookFor("button");
  crawlActions.lookFor("input").withAttribute("type",
                                "submit");
  crawlActions.lookFor("input").withAttribute("type",
                                "button");
}
```

Figure 1: Example of inconsistent method name in code evolution.

*Meng Yan is the corresponding author

[1]https://github.com/crawljax/crawljax

979-8-3503-1958-3/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00074

proposed to identify inconsistent method names or recommend appropriate names. However, we find that the current researches are not a good solution to the task of JIT Method Name Updating, as they do not take into account the information from the previous version of the method, which could provide important insights for updating the method name. Moreover, there is a lack of a specific dataset for this task at present. Therefore, we construct a dataset with over 108K method name-body co-change samples, which are extracted from popular Java projects hosted on GitHub. To investigate this task, we conduct an empirical analysis of the dataset that we constructed to study the characteristics of method name updates. Based on our findings that method name updates involve a significant number of single body token change samples which includes a substantial portion of code-indicative updates, we propose the use of heuristic rules to handle such samples. For other changing samples, we build a customized seq2seq model with some customization to learn the pattern of method name updating and better fit the task at hand. We integrate these two components to form an approach that combines heuristic rules and a neural model to handle this task better.

To assess the effectiveness of our approach, numerous experiments are conducted on our dataset. We compare our approach, MAP (**M**ethod n**A**me u**P**dating), against three baselines: a commonly used baseline in the JIT task (Origin [38]), a method based on context information and prior knowledge (Cognac [46]), and a general neural machine translation model (NMT). The evaluation is conducted on our dataset, considering metrics of Accuracy, Precision, Recall, F1-score, Average Edit Distance (AED), and Relative Edit Distance (RED) following prior studies [33], [34], [37], [38]. The evaluation results demonstrate the superior performance of MAP over all baselines across all metrics. Moreover, the ablation study reveals that all the design components contribute to the performance of MAP in method name updating. Noticeably, the utilization of heuristic rules and token type stand out as the two most influential factors in achieving improved results.

In summary, this paper makes the following contributions:

- We first propose the task of JIT Method Name Updating and construct a dataset with more than 108K method name-body co-change samples, which represents the first extensive dataset available for this task.
- We designed the approach, namely MAP (**M**ethod n**A**me u**P**dating), which is based on the hybrid of the seq2seq neural network and heuristic rules, and could effectively address the unique characteristics of this task.
- We conduct a comprehensive evaluation of the dataset. The evaluation results indicate that our approach is highly effective and significantly outperforms the baselines on the task of JIT Method Name Updating.
- We have made our replication package public[2], including the dataset, the source code, our trained model, and test results. This enables other researchers to easily replicate our work and build upon new findings in future studies.

[2]https://anonymous.4open.science/r/MNU-139/

2. Related Work

2.1 Method Name Recommendation

The task of method name recommendation (MNR) aims to generate method names automatically with high quality to guarantee the legibility of source code [12], [19]. Numerous approaches have been suggested to tackle it. Liu et al. [36] follow an information retrieval (IR) direction with the key idea that two methods with similar bodies should have similar names. Code2vec [8] and Code2seq [7] are two neural network models that represent a method body as a distributed vector by aggregating the bag of Abstract Syntax Tree (AST) paths with the attention mechanism. They suggest reusing the names of methods that share similar AST structures with the target method. MNire [39] uses the sub-tokens in the program entities' names. Qu et al. [41] build a code graph called code relation graph (CRG) to describe the code structure and utilize the graph neural network models [6], [30]to encode it, and then combined it with the code token sequence embedding to generate new method name through RNN [24]. Wang et al. [46] propose a context-guided method name recommender that leverages both local and global contexts to generate method names, with empirical evidence as the guiding prior knowledge. Liu et al. [35] otherwise consider the project-specific context and documentation of the target method. Li et al. [31] develop DeepName, a context-based approach that extracts the features from four contexts: the internal context, the caller and callee contexts, sibling context, and enclosing context.

2.2 Method Name Consistency Checking

The task of method name consistency checking (MCC) is to automatically check the consistency between method name and source code to avoid misunderstanding and program defects [3], [4], [9]–[11], [40], [48]. Hùst and Østvold [26] utilize the Java language naming convention to extract rules for method names, which are then used to identify naming bugs. Kim et al. [27] construct a code dictionary from existing API documents and used it to detect inconsistent names. Allamanis et al. [5] propose to learn the domain-specific naming convention from local contexts to enhance stylistic consistency, including identifier naming and formatting. Liu et al. [36] separately encode method names and implementations and consider two sets of method names to compare the similarity of these two sets. The work of Son [39] and Wang et al. [46] can also be utilized for the MCC task by comparing the similarity between the recommended name and the original name of the method.

2.3 Just-In-Time Task

During the code evolution process, defects and inconsistencies may be introduced due to the code changes [13], [22]. The just-in-time (JIT) task is to automatically identify and update these undesirable issues. Specifically, Hoang et al. [23] introduce a comprehensive deep learning framework that leverages commit messages and code changes to solve the JIT defect prediction task. Liu et al. [37], [38] otherwise, propose the task of JIT comment detection and updating, and developed a novel Seq2seq model to address this task. Lin et al. [33] suggest a

rule-based approach based on extensive empirical analysis to efficiently address it. These works demonstrate the significance of JIT tasks and highlight the need to address and resolve such issues, which also serves as inspiration for our research.

3. Empirical Study

To explore the task of the JIT Method Name Updating, we conduct empirical studies on the constructed dataset, which is obtained from the commit history of a number of Java projects. In this section, we would first introduce the dataset preparation, and then try to provide answers to the following research questions:

RQ1: What is the distribution of the edit operation in the task of method name updating?

RQ2: What is the connection between the modified tokens and the new method name?

3.1 Data Preparation

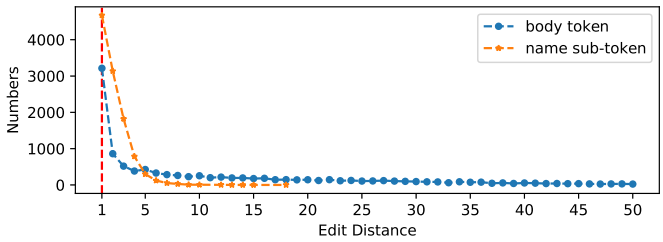
The dataset constructed in this study is derived from real data collected from the GitHub repository. Specifically, we use the 1,496 collected repositories from the work of Liu et al. [38]. These repositories have undergone careful filtering by Liu et al. [38], thus could alleviate the potential biases of the constructed dataset. The method body and corresponding method name that underwent modifications were extracted from non-merge commits of each repository to construct method name-body co-change instances. To do this, we used the GumTree [18] matching algorithm, which was customized by Liu et al. [38], to calculate the method mapping between the new and old versions of the code. Next, we compared the Abstract Syntax Tree (AST) of each old and new version method based on the mapping to identify and extract the method code that underwent modifications.

To ensure a more robust dataset, we filtered out unsuitable instances such as abstract methods, empty method bodies, methods whose name is changed while bodies are not, and duplicate data to prevent data leakage. To reduce training time, we set the maximum length of the method body information, old method name, and new method name to 500, 20, and 20, respectively, based on the 90% quantiles of the dataset, ensuring that most methods were included within this length without causing training device issues. Ultimately, we obtained 108,360 method name-body change instances.

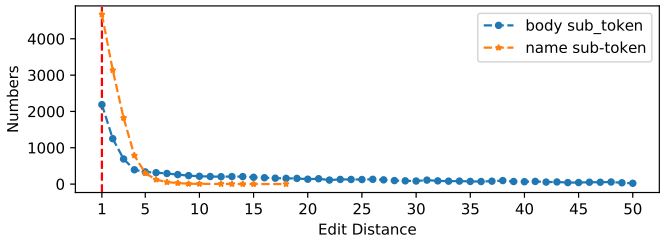
During the process of dividing the dataset, the instances were sorted in ascending order based on the submission creation time of each project. The first 80% of the instances were assigned to the training set, and the remaining 20% were randomly shuffled and equally assigned to the validation and test sets, ensuring that all method names in the training set occurred before those in the validation and test sets, avoiding data leakage that could exaggerate method performance. Thus, the final training, validation, and test sets consisted of 86609, 10805, and 10946 method body-method name co-change instances, respectively.

3.2 Edit Distance Statistics

The main feature of the JIT Method Name Updating task is that it takes into account the modification properties of the method body. To gain insight into this task, we count the edit distance between the pre-modified and post-modified method name(body) in our dataset. Given two versions of method names(bodies), edit distance is the minimum edit tokens (insert, delete, and replace) required to change a method name(body) into the other, and calculated the same as word-level Levenshtein distance [29]. First, we need to mark and align the pre-modified and post-modified methods, which will be described in detail in Section 4-A2. Then we count the numbers of the edit which is not "equal" to obtain the edit distance. Two levels of statistics are designed, namely the token level and the sub-token level (for compound words), to explore the change from a microscopic perspective.



(a) Edit distance comparison between method body token and method name sub-token.



(b) Edit distance comparison between method body sub-token and method name sub-token.

Figure 2: Statistics of edit distance.

Figure 2 shows the statistics of the edit distance for method name and method body changes. It can be seen that the distribution of the edit distance is quite similar compared to the method name or the method body, as well as the token level and sub-token level. The majority of method bodies and method names exhibit a small edit distance. As the edit distance increases, the number of edit distance gradually decreases. Specifically, the edit distance of the method name decreases gradually from 0 to 10, and the token (sub-token) edit distance for the method body is mainly concentrated in the range of 0 to 4. Notably, the most frequent edit distance is 1, approximately 42.3% in method name, 29.4% for token level, and 20.0% for sub-token level in method body, which suggests that there may exist a one-to-one correspondence between modifications of the method body and method name.

> **Finding-1:** The distribution of edit distance for method names and method bodies is consistent. As the edit distance increases, the number of both method bodies and method names gradually decreases. Among all edits, the single modification is the most frequent.

Code-Indicative Sample

Before Editing:

```
protected Timestamp getDocumentDat(){
    ...
    return documentDate;
}
```

After Editing:

```
protected Timestamp getDateDoc(){
    ...
    return dateDoc;
}
```

Non-Code-Indicative Sample

Before Editing:

```
public URL sanitizeURL(URL url){
    ...
    return url;
}
```

After Editing:

```
public Document getFirstPage(){
    ...
    return Http.url(this.url).get();
}
```

Figure 3: Example of code-indicative update, and non-code-indicative update.

TABLE I: Proportion of code-indicative sample in the test dataset

test dataset	U	CIU	Proportion(%)
all	10946	5928	54.17
single token	3214	2620	81.52
single sub-token	2188	1809	82.68

*U represents the samples that need to be updated; CIU represents the code-indicative update sample.

3.3 Modified Tokens Statistics

According to the previous work [33], [34], we divided the update of the method name into two categories: code-indicative update and non-code-indicative update. A formal definition of this classification standard is as below: an instance of method name update is classified as a code-indicative update if all of its updated contents can be found in the corresponding code change, and non-code-indicative update otherwise [33]. Figure 3 shows two specific examples.

To further investigate whether the update of method name could be promoted by code-indicative information, we count the sample proportion of code-indicative update in all the test samples, single token update samples (edit distance of token level is 1), single sub-token update samples (edit distance of sub-token level is 1). The outcomes are presented in Table I. Through the table, we could find that the code-indicative update sample accounts for half of the proportion in all test samples, and the percentage of code-indicative update samples for a single token or single sub-token update is even higher. 2620 of the 3214 single token change samples are code-indicative updates, which is more than 80%.

Specifically, code-indicative updates involve cases where identifier renaming or function call substitution occurs in the source code, and the renamed object is key content in the method name. As the example of code-indicative update in Figure 3, "documentDate" is modified to "dateDoc", and the updated content is consistent with the method name. These code-indicative updates could be more easily updated by heuristic-based rules, allowing for accurate updates to be made [33].

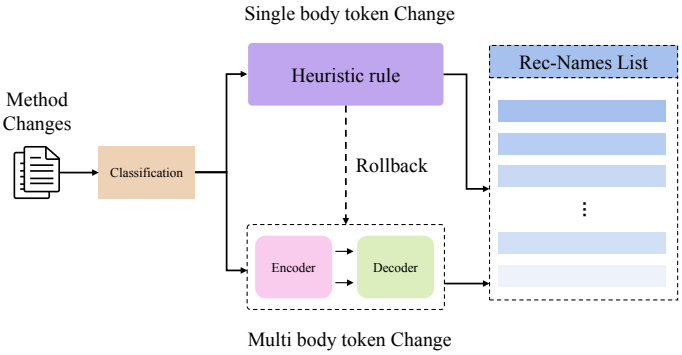


Figure 4: Overview of our approach.

> **Finding-2:** Code-indicative updates are prone to appear in single token change samples, which is also the most frequent edit in Finding-1. This inspires us to use heuristic rules to update them.

4. Approach

Figure 4 illustrates the overall framework of our approach, which involves three components: classification, heuristic rule-based component, and neural model-based component. The detailed steps of our approach are as follows:

1) **Data Construction:** Given the method body changes and the old method name of the code, our method first tokenizes the method body and method name, and adds the corresponding editing operations and token types to each token of the separated method body to facilitate the updating process.
2) **Sample Classification:** Based on the body content of the modifications, we classify the samples into **single body token change samples** (edit distance equals to 1) and **multi body token change samples** (edit distance greater than 1).
3) **Updates Generating:** The single body token change samples will be directly addressed by the heuristic rule-based component, while the multi body token change samples will be solved by the neural model-based component.
4) **Rollback:** If the heuristic rules could not generate any update for single body token change samples, we switch to the neural model to update the method name.

In this section, we first describe how to get the edit representation of changes, and then explain how to separate single body token change samples and multi body token change samples. The detailed process of the heuristic rule-based component and the neural model-based component are then illustrated.

4.1 Data Flatting

4.1.1 Separation

First, we obtain two versions of the method (the snippet before the modification and the snippet after the modification). Then,

we employ Javalang [2] tool to parse these two methods so that the method names can be extracted. After that, these two snippets are split into $< n_{src}, b_{src}, n_{dst}, b_{dst} >$ where n represents the method name and b represents the method body. In cases where parsing errors occurred while adopting Javalang tool, we would apply rules based on regular expressions to collect the method names.

4.1.2 Tokenization

After separating the method name and method body, we also need to tokenize them to meet the requirements of our approach. The tokenization of the method body includes both token level and sub-token level, while the method name is directly tokenized at sub-token level. **1)Token level tokenization for method body:** The method bodies of the two snippets are tokenized at token level using a lexer [38]. This lexical analysis is performed with the help of a highly specialized tool that is designed to identify and extract individual tokens from the source code. During this process, the lexer not only separates the code into individual tokens but also removes any inner comments or extraneous white spaces that may be present in the code. **2)Sub-Token level tokenization for method body:** Compound words (e.g., buttonGreenPass), constructed by concatenating multiple vocabulary words according to camel or snake conventions, appear frequently in the code. We split them into multiple sub-tokens to reduce the number of out-of-vocabulary (OOV) words following previous studies [7], [46]. The sub-tokens are joined with the "$< con >$" token to mark that two adjacent tokens were originally concatenated [38]. **3)Tokenization for method name:** Method names are tokenized just like compound words because method names are essentially composed of tokens (sub-tokens) in camel or snake conventions. Now, we get $< T_{n_{src}}, T_{b_{src}}, T_{n_{dst}}, T_{b_{dst}} >$, where $T = [t_1, t_2, t_3, ..., t_{|T|}]$ and t_i is a sub-token of method name(method body).

4.1.3 Alignment

After the method bodies of the two code snippets are tokenized, we align the two generated token sequences to accurately capture the changes that have occurred. To achieve this, we use a complex diff tool called SequenceMatcher [1], and a set of heuristics [38] that can construct an accurate alignment sequence. By aligning the token sequences in this way, we are able to effectively represent each code change in a much more granular and detailed manner. This is particularly important when dealing with complex software code bases, where even minor modifications can have far-reaching implications.

4.1.4 Code Change Representation

After alignment, each element in the sequence would be added with an extra item named an *edit* to construct an edit sequence. Every element in the edit sequence is a triple $< t_i, t_i^{'}, a_i >$, where t_i is a token in the old code, $t_i^{'}$ is a token in the new code, and a_i is the edit action that transforms t_i to $t_i^{'}$, which consists of 4 types (i.e., insert, delete, equal, and replace). If a_i is insert (delete), then t_i ($t_i^{'}$) is represented by the empty token $\varnothing$. Edit sequences preserve the information of both old and new code versions and highlight the fine-grained changes between them, which is similar to the approach in [49]. By capturing these fine-grained modifications between the old and new code versions, we are better equipped to analyze and understand the changes that have occurred, and to make informed decisions about how best to update method names.

TABLE II: Method body statement types and their labels

Statement Type	*label*	Statement Type	*label*
ExpressionStatement	1	SwitchCase	9
LocalVariableDeclaration	2	ReturnStatement	10
AssertStatement	3	DoStatement	11
WhileStatement	4	ForStatement	12
IfStatement	5	FieldDeclaration	13
TryStatement	6	SynchronizedStatement	14
ThrowStatement	7	UNK	15
SwitchStatement	8	Other	0

4.1.5 Token Type Representation

According to the empirical study of wang et al. [46], token types contribute differently to the composition of a method name. That is, the information of different code statements can be utilized to better guide the generation of method names, of which Return Statement has the highest contribution, followed by Expression Statement and Try Statement last. However, we do not directly use the probabilities from Wang et al.'s work as prior knowledge because the dataset they used for empirical analysis is different from the dataset used in this study, which may result in a bias in token type probabilities. Instead, we provide types of tokens that allow the model to learn the priority of different tokens through training, thus providing more accurate guidance for updating method names in the task.

We use Javalang tool to parse the AST structure information of the method so that each token would get a type from the result, and we label all the types, which are connected to the back of the edit operation. The final goal of the JIT Method Name Updating task is to generate the name of the latest method body code, so we only extract the token type from the post-modified method body to prevent interference from the information of the pre-modified code snippet. We adopt the statement types proposed by Wang et al. [46] and define their corresponding label as shown in Table II.

After stitching the token types together, we get a quadruplet $< t_i, t_i^{'}, a_i, l_i >$, where the l_i is the type label of a token $t_i^{'}$. So, the input of our method can be expressed as $E_{edit} = [< t_1, t_1^{'}, a_1, l_1 >, < t_2, t_2^{'}, a_2, l_2 >, ..., < t_n, t_n^{'}, a_n, l_n >]$.

4.2 Sample Classification

Based on empirical analysis, we know that code snippets with single body token changes contain more code-indicative modifications, which are easy to be updated by heuristic rules. To handle such single body token change samples by rule-based component and employ neural model-based component for multi body token change samples, we use the following approach to determine the sample types: After

the method body of the two versions is processed through step Tokenization(4-A2) and Alignment(4-A3), the code body change is segmented into edit sequence. By finding the quadruplet elements, that is, quadruplet elements that are not *equal* in edit sequence. If only one quadruplet is not *"equal"* in a sample, i.e., edit distance equals 1, it is classified as a single body token change sample.

4.3 Heuristic Rule Based Component

The content of this section is based on the findings in Section 3. In a large number of modifications, there is a high proportion of single-token modifications, and a high proportion of these are code-indicative modifications. Therefore, we designed a heuristic rule for single-token modification code, which is shown in Figure 5. The steps of this rule include 3 steps: 1) locating the change token in the method body, 2) constructing the replacement pairs, and 3) matching the updates.

4.3.1 Locating the Change Token

After the classification step in Section 4-B, the single token update samples will be identified. The heuristic rule only needs to process the modified tokens, so it is necessary to locate the modified token and extract it separately. Meanwhile, we removed the connecting element "$< con >$" and no longer considered the connection relationship, as we know that such sub-tokens are definitely continuous if only one token is modified.

4.3.2 Constructing the Replacement Pairs

Next, we will construct token-level replacement pairs for updating method names. It is accomplished by considering all possible modifications, which consist of sub-token sequences from the modified token. Each pair of aligned sub-token sequences represents a potential update that can be applied. Specifically, when given two aligned sub-token sequences A and B, we can construct a replacement pair $< A[i:j], B[i:j] >$, where $A[i:j]$ represents sub-tokens i to j in sequence A, and $B[i:j]$ represents the corresponding sub-tokens in sequence B as shown in Figure 5 (b). When sub-tokens are concatenated to form a token, any instances of the $\emptyset$ placeholder are simply ignored.

4.3.3 Matching the Updates

After constructing the replacement pairs, we search for any token sub-sequences in the pre-modified method name that match the replacement pairs. If a match is found, we use the corresponding new tokens to update the method name and generate the predicted method name. Specifically, we would traverse the set of replacement pairs and check if each old token sub-sequence exists in the pre-modified method name. If so, we update the old tokens with the corresponding new tokens and use the updated method name as the predicted method name. When multiple updates occur, we generate a candidate list and arrange it in descending order based on the number of sub-tokens they own. If no matching tokens are found after traversing the replacement pairs, we would otherwise update the method name using the model-based component.

4.4 Neural Model Based Component

This section provides a neural network-based model, whose architecture is shown in Figure 6.

4.4.1 Embedding Layer

The edit embedding layer takes the edit sequence $E_{edit} = [< t_1, t_1^{'}, a_1, l_1 >, < t_2, t_2^{'}, a_2, l_2 >, ..., < t_n, t_n^{'}, a_n, l_n >]$ and the old name sub-token sequence $E_{src_{name}} = [x_1, x_2, ..., x_m]$ as input, mapping them into feature vectors so that they are available for neural model.

The embedding methods for different types of elements vary slightly. 1) For the new and old method body and old method name, a shared vocabulary is constructed. This shared vocabulary maps the tokens in the method body and method name into a shared vector space to ensure consistent embedding of the same tokens in both components, thereby simplifying information capture between the method body and the method name. Then, the pre-trained model (e.g., FastText [20]) is used to obtain word embedding for each token, shown in (1), as they utilize a large amount of code corpus for training, thereby providing accurate syntactic and semantic information of code representation. 2) For the editing operation a_i and the types of new method body token l_i, we initialize their embedding matrices randomly and continuously update them during training to allow the model to learn their weights freely as shown in (2). In this way, the model would learn to represent the token *edits* and token *types* in their respective vector spaces.

$$e_{t_i}, e_{t_i'}, e_{x_i} = Emb_{pre}(t_i), Emb_{pre}(t_i^{'}), Emb_{pre}(x_i) \quad (1)$$

$$e_{a_i}, e_{l_i} = Emb_{rand}(a_i), Emb_{rand}(l_i) \quad (2)$$

4.4.2 Contextual Embed Layer

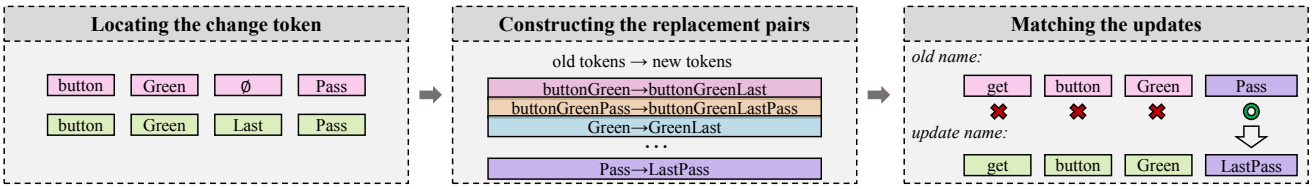
In this layer, we use two separate bidirectional LSTM (Bi-LSTM) [24] layers to further learn the information of the edit sequence and the old method name, and transform them into context vectors. For edit sequence, the input embeddings, i.e., the quadruplet $< e_{t_i}, e_{t_i'}, e_{a_i}, e_{l_i} >$, of each edit E_i are first concatenated horizontally, and fed into the Bi-LSTM. And for each embedding e_{x_i} of old name sequence $E_{src_{name}}$, is directly input into another Bi-LSTM. The formulas for calculating the context vectors $h_i^{'}$ for the method body and h_i for the old method name information are given by equations (3) and (4) respectively.

$$h_i^{'} = Bi\text{-}LSTM(h_{i-1}^{'}, h_{i+1}^{'}, [e_{t_i}; e_{t_i'}; e_{a_i}; e_{l_i}]) \quad (3)$$

$$h_i = Bi\text{-}LSTM(h_{i-1}, h_{i+1}, e_{x_i}) \quad (4)$$

4.4.3 Attention Layer

To capture the relationships between the body change and the old name, we use an attention layer shared by the two which could link and fuse their information. The layer takes as input the contextual vectors $H^{'} = [h_1^{'}, h_2^{'}, ..., h_n^{'}]$ and $H = [h_1, h_2, ..., h_m]$, which represent the body change and old name respectively. The output is a feature vector $g_i^{'}(g_i)$ for



(a) (b) (c)

Figure 5: Heuristic rule-based component.

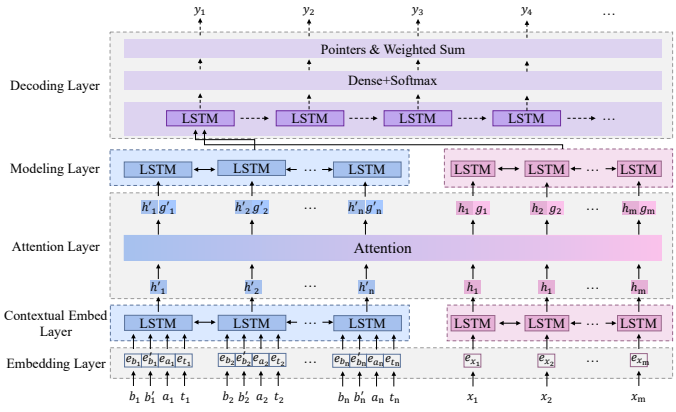


Figure 6: Neural model-based component.

each $h_i^{'}(h_i)$, as well as the original contextual $h_i^{'}(h_i)$ vector of that edit (name), which is passed to the next layer. Specifically, each $g_i^{'}(g_i)$ feature vector will be computed through dot product attention mechanism [45]. The formula for the feature vector $g_i^{'}$ is shown in (5) and (6).

$$\alpha_i^{'} = softmax(H^T W_\alpha^T h_i^{'}) \tag{5}$$

$$g_i^{'} = H\alpha_i^{'} \tag{6}$$

where $\alpha_i^{'}$ represents the attention weights that measures how important each x_i respect to edit. W_α^T is the trainable parameters.

And the calculating of g_i is shown in (7) and (8). Where α_i is also the attention weight but represents the importance of each edit with respect to x_i.

$$\alpha_i = softmax(H^{'T} W_\alpha h_i) \tag{7}$$

$$g_i = H^{'} \alpha_i \tag{8}$$

4.4.4 Modeling Layer

This layer consists of two different Bi-LSTM models, which are used to learn the final representations of each method body edit sequence and old method name sub-token sequence. Specifically, given an old method name token x_i, its final representation u_i is calculated based on the context vector h_i and the attention feature vector g_i, as shown in (9). The final representation $u_i^{'}$ of the method body edit information sequence is computed in the same way, which is shown in (10).

$$u_i = Bi\text{-}LSTM(u_{i-1}, u_{i+1}, [g_i; h_i]) \tag{9}$$

$$u_i^{'} = Bi\text{-}LSTM(u_{i-1}^{'}, u_{i+1}^{'}, [g_i^{'}; h_i^{'}]) \tag{10}$$

4.4.5 Decoding Layer

The decoding layer, which consists of three main parts: a unidirectional LSTM, an attention-based linear layer, and a weighted sum layer based on pointer copying, ultimately generates a new method name.

LSTM Layer. The LSTM layer generates new annotations by sequentially generating their tokens, taking the feature matrices $U^{'} = [u_1^{'}, u_2^{'}, ..., u_n^{'}]$ and $U = [u_1, u_2, ..., u_m]$ output from the encoder as input. To construct the initial state s_0 of the LSTM layer, the last feature vectors of U and U' are concatenated in series. For each decoding step j_i, we consider the policy of teacher forcing. The ground truth input $\hat{y_j}$ is first mapped to a feature vector $e_{\hat{y_j}}$ using the same embedding layer as old method name in Embedding Layer (4-D1). Here, $\hat{y_j}$ is the reference token from the previous time step during training, i.e., the ground truth token of the new method name. While during testing, it is the token generated by the previous time step. Then, the hidden state s_j of this step is computed as (11), where s_{j-1} is the previous hidden state and o_{j-1} is the previous output vector.

$$s_j = LSTM(s_{j-1}, [e_{\hat{y_j}}; o_{j-1}]) \tag{11}$$

Attention-based Linear Layer. Dot-product attention is also applied in the decoding process. At each decoding step, the context vectors c_j and $c_j^{'}$ from the old method name and method body information (both calculated like (5) and (6)), as well as the current decoding state s_j, are concatenated to calculate the vocabulary distribution p_j^{vocab} as below:

$$o_j = tanh(V_c[c_j; c_j^{'}; s_j]) \tag{12}$$

$$p_j^{vocab} = softmax(V_c^{'} o_j) \tag{13}$$

where V and $V^{'}$ are trainable parameters.

Weighted Sum Layer. The pointer network [43] is applied to alleviate the problem of out-of-vocabulary (OOV) words. We use two pointer generation networks to copy tokens from the old method name and the new method body, respectively. The calculation formulas for the two pointer generation networks are shown in equations (14) and (15).

$$p_j^{name} = \sum_{k:x_k=y_j} a_{jk} \tag{14}$$

$$p_j^{vocab} = \sum_{k:b_k=y_j} a_{jk}^{'} \tag{15}$$

$P_j^{name}(y_j)$ and $P_j^{body}(y_j)$ are the probabilities of copying tokens from the old method name and the new method body, respectively. a_{jk} and $a^{'}_{jk}$ are the attention weights. Finally, the conditional probability of generating target token y_j at time step j is calculated as:

$$\begin{aligned} p(y_j|y_{<j}, x, E) =& \gamma_j P_j^{vocab}(y_j) + (1-\gamma_j)* \\ & (\theta_j P_j^{name}(y_j) + (1-\theta_j) P_j^{body}(y_j)) \end{aligned} \tag{16}$$

where, γ_j, θ_j, and $1-\theta_j$ are the probability of selecting y_j from the vocabulary, the old method name, and the new method body, respectively. They are all trained with the decoder.

5. Evaluation Setup

5.1 Experiment Settings

The MAP model is implemented in Python version 3.6.12 and employs the PyTorch deep learning framework, version 1.2.0, on a server equipped with NVIDIA GeForce RTX 3090 GPUs. The unified vocabulary created retains only tokens that appear more than once, resulting in a size of 980KB. The editing actions, new method body types, new and old method body tokens, and old method name tokens are embedded in 300-dimensional vectors. The Bi-LSTM and LSTM hidden states are 256 and 512 dimensions, respectively. The random dropout rate for all LSTM and linear layers is set to 0.2.

During training, cross-entropy minimization is used as the objective, and the Adam gradient descent algorithm is employed to optimize the model with a learning rate of 0.001 and a gradient clipping size of 5. During validation, the perplexity is computed every 500 batches on a validation set with a batch size of 32. If the perplexity does not decrease after five validations, the learning rate is reduced by half and flagged. If the number of flags reaches five, training is stopped, and the model with the best validation score is utilized for testing. During testing, a Beam Search method with a width of 5 was applied to generate the final method name. In order to reduce the impact of random errors, the MAP model was trained and evaluated 10 times, with the average performance being taken as the evaluation result.

5.2 Baselines

As we perform the first exploration of the "Just-In-Time Method Name Updating" task, there are no existing methods available for this task to compare. In order to evaluate the performance, we select three different benchmarks: Origin, Cognac, and NMT. Brief introductions of these methods are presented below:

Origin: It refers to a baseline that produces the previous name as its output. In the just-in-time task, it is commonly used to verify whether the generated result is closer to the new version than the old one [38].

Cognac: It is a Seq2seq model based on context information and prior knowledge proposed by Wang et al. [46], which focuses on two aspects of information and performs well in MNR task. It extracts the context of itself and its callers or callees method to introduce more useful information and utilizes the empirical observation as the prior knowledge to pay more attention to key tokens. The above two parts of information are taken as input to generate method names based on Seq2seq model consisting of a Bi-LSTM. By comparing with this method, we can investigate whether updating problem is significant.

NMT: Neural Machine Translation model, utilizing a generic neural sequence-to-sequence (seq2seq) structure, is commonly used as a baseline for the generation task. We build the model based on the LSTM with an encoder-decoder structure which is similar to our neural method component. The attention mechanism is also used to enhance the model's ability to capture correlations between the decoders and encoders. By contrasting our proposed approach with the NMT, we can investigate whether our method is superior in performance for the universal generative model.

5.3 Variants of MAP

To analyze the impact of individual components on the performance of our approach, we have created several variants, namely MAP-NoFt, MAP-NoUni, MAP-NoAttn, MAP-NoType, and MAP-NoHeb.

- **MAP-NoFt** does not use the pre-trained model FastText.
- **MAP-NoUni** uses two distinct vocabularies, instead of a unified one, for method body and method name tokens.
- **MAP-NoAttn** removes the attention layer and keeps other components the same.
- **MAP-NoType** does not consider token type information for the new method bodies.
- **MAP-NoHeb** eliminates the heuristic rule-based component and only a neural network model is used.

5.4 Evaluation Metrics

In order to validate the performance of our approach, we employ 6 evaluation metrics, namely Accuracy, Precision, Recall, F1-score, AED, and RED.

Accuracy is a method-level metric used to measure the ability to generate correct method names. Specifically, if the generated method name is identical in order and content to the ground truth method name, it is considered correct. Otherwise, it is considered incorrect.

Precision, Recall, and F1-score are token-level metrics used to evaluate the extent to which the model can generate correct method name tokens, i.e., the proportion of correct method name tokens that are generated. For a correct method name token o, and a generated method name token r, the formulas for these three evaluation metrics are shown in (17), (18), and (19), respectively.

$$precision(r, o) = \frac{|token(r) \cap token(o)|}{|token(r)|} \tag{17}$$

$$recall(r, o) = \frac{|token(r) \cap token(o)|}{|token(o)|} \tag{18}$$

$$F1\text{-}score(r, o) = \frac{2 * precision * recall}{precision + recall} \tag{19}$$

TABLE III: Comparisons of our approach with each baseline

Approach	Acc.	Pre.	Rec.	F1	AED	RED
Origin	0%	52.7%	51.6%	52.1%	2.068	1
Cognac	24.2%	61.0%	52.0%	56.2%	1.921	1.161
NMT	43.5%	74.6%	72.0%	73.3%	1.172	0.595
MAP	**47.7%**	**76.2%**	**73.1%**	**74.6%**	**1.111**	**0.556**

Average Edit Distance (AED) and Relative Edit Distance (RED) are updating metrics used to measure the edit operation of the generated name. AED calculates the average number of edits developers need to make in order to achieve perfect updates after using an updater. Similarly, the RED metric measures the average relative edit distance. Specifically, for a given test set with N samples, the AED and RED of an approach are defined as follows:

$$AED = \frac{1}{N}\sum_{k=1}^{N} edit\_distance(\hat{y}^{(k)}, y^k) \tag{20}$$

$$RED = \frac{1}{N}\sum_{k=1}^{N} \frac{edit\_distance(\hat{y}^{(k)}, y^k)}{edit\_distance(x^{(k)}, y^k)} \tag{21}$$

where $\hat{y}^{(k)}$ is the generated method name, $x^{(k)}$ is the old method name, and y^k is the reference method name.

6. Evaluation Results

6.1 RQ3: Effectiveness of Our Approach

To explore the effectiveness of our approach, i.e., MAP, we compare it with three baseline methods on our dataset in terms of Accuracy, Precision, Recall, F1-score, AED, and RED. The results are presented in Table III.

From the table, we can observe that Origin achieves the worst performance across all the metrics. Given a modified/new method body, Origin directly returns the old method name as the expected result. Obviously, the old method name can not be adaptable to the new method body, resulting in an accuracy value of 0.00%. Interestingly, despite the poor accuracy, the recall of Origin is still more than 50%, which suggests a significant overlap between the new and old method names for the task of JIT Method Name Updating. This indicates that the old method name still contains some valuable and reusable tokens that appear in the new method name. Therefore, it is reasonable for us to use the old method name as a template instead of generating method name from scratch.

For Cognac, the improvement of recall is only 0.77% compared with Origin. This is because Cognac needs to generate tokens one by one from scratch. As a classical Seq2seq model, Cognac fails to generate low-frequency words, such as project-specific identifiers, and is difficult to handle the method name with a long sequence. Additionally, Cognac, as a static method name generation technique, struggles to understand and focus on the modified token between new and old method bodies. However, such modified tokens may contribute the most to the new method of name generation. NMT utilizes the learned update patterns for editing old method names and achieves better results than Cognac. However, We can see that MAP outperforms all the baselines in terms of all evaluation metrics. The accuracy, precision, recall, F1-score of MAP are 47.7%, 76.2%, 73.1%, and 74.6%, with the improvement of 9.66%, 2.14%, 1.53%, and 1.77%, compared with NMT. Considering the metrics of AED and RED, MAP significantly outperforms both Origin and Cognac, reducing the relative edit operations by nearly 50%. These results indicate MAP can update names more effectively and accurately than the three baselines.

> **Overall:** MAP outperforms the three baselines on all metrics, demonstrating the effectiveness of our method on the JIT Method Name Updating task.

TABLE IV: Performance of variants

Approach	Acc.	↓	Pre.	↓	Rec.	↓	F1	↓
MAP-NoFt	44.1%	3.6%	74.8%	1.4%	71.9%	1.2%	73.3%	1.3%
MAP-NoUni	44.5%	3.2%	75.2%	1.0%	71.9%	1.2%	73.5%	1.1%
MAP-NoAttn	43.9%	3.8%	74.8%	1.4%	72.0%	1.1%	73.4%	1.2%
MAP-NoType	43.5%	**4.2%**	74.6%	**1.6%**	72.0%	**1.1%**	73.3%	**1.3%**
MAP-NoHeb	45.9%	1.8%	75.8%	0.4%	72.5%	0.6%	74.1%	0.5%
MAP	**47.7%**		**76.2%**		**73.1%**		**74.6%**	

↓ denotes performance degradation relative to MAP.

6.2 RQ4: Contributions of Main Components

In this research question, we evaluate and compare MAP with its variants to provide valuable insights into the impact of these components on performance. The experimental results are presented in Table IV.

Upon analyzing the performance metrics, we can see that our approach exhibits the best results than the other variants. When removing any component, the performance of MAP degrades to varying degrees. For instance, removing the attention layer (i.e., MAP-NoAttn) leads to a 3.8% and 1.2% decrease in Accuracy and F1-score, respectively. Notably, among all variants, the Accuracy difference between MAP and MAP-NoType is the largest. When we remove the token type of the input, the effect will drop by 4.2% in Accuracy and 1.3% in F1-score. This observation highlights the significance of incorporating token type information in JIT Method Name Updtaing. The inclusion of type information may enable the neural model-based component to prioritize specific token types during training, ultimately facilitating improved performance.

At the same time, it can be observed that the heuristic rule-based component is also indispensable for the overall method. While the results only show a slight decrease in Accuracy, Precision, Recall, and F1-score, it still contributes to the model as a whole. This could be attributed to the model's potential errors or omissions in generating certain tokens when it comes to single-token updating, resulting in lower performance compared to the heuristic rule-based approach.

> **Overall:** These results indicate that all the main components are useful and combining all components is the best.

Method Body Changes

Before Editing:

```
{
   ...
   return innerVarPatterns();
}
```

After Editing:

```
{
   ...
   return innerStatements();
}
```

Origin: “implicit”,“inner”,“var”,“patterns”
Cognac: “statements”
NMT: “implicit”,“inner”
MAP: “implicit”,“inner”,“statements”
Ref: “implicit”,“inner”,“statements”

Figure 7: Example of updating.

7. Discussion

7.1 Do Our Task and Approach Make Sense

Previous research has shown that issues of inconsistency due to modifications are pervasive, and the best approach is to address them before introducing the code into the software code bases [23], [33], [38], [50]. As an example in Figure 7, the function call in the method body has been changed from “inner**VarPatterns**()” to “inner**Statements**()”. However, due to oversight or forgetfulness, the developer does not update the method name, resulting in an inconsistent method name “implicitInnerVarPatterns” with the new method body. If such source code is brought into the repository, it is highly probable to cause a series of chain reactions, such as misusing and introducing defects. Additionally, it would increase the time and manpower costs required for code repository maintenance and fixes. This indicates that the task of JIT Method Name Updating, which aims to detect and rectify method name inconsistencies, is highly relevant and significant in minimizing the potential introduction of such issues.

To investigate the effectiveness of our approach in JIT Method Name Updating, we manually examined several examples. For the sample in Figure 7, Cognac generates one relevant token as a recommendation. However, it lacks the token “implicit”, which is derived from the previous version. Without this token, the semantic information provided by programmers may potentially be lost, complicating the understanding of the source code. As for NMT, it contains old information but lacks the crucial token “statements” during generation. This may be due to NMT’s failure to capture important modified tokens without token type which is special in the software engineering field, resulting in a missed generation. Furthermore, NMT, being a pure neural model, has inherent randomness and a black-box nature. Our proposed approach, on the other hand, combines heuristic rules and a neural model that adds token-type information and can reduce the occurrence of missing and erroneous generation to a certain extent, resulting in accurate updates.

7.2 Threats to Validity

A potential limitation to the effectiveness of this work is that our dataset is constructed exclusively from Java projects, which may not be representative of all programming languages. Different languages may exhibit slight variations in change patterns. Therefore, our methods may not be fully applicable to other languages, and this will be further explored in our future work. Another threat is the inability to guarantee that all methods in our data have high-quality method names for 1) the names of the methods in the before and after edit versions may not always be consistent; 2) method name updates during the evolution process may not always be driven solely by changes in the method body. For example, the personal programming preferences of developers can also play a role. To address it, we build a dataset composed of well-maintained open-source projects. Furthermore, such noise is acceptable for deep learning techniques, we think the impact of this issue is limited.

7.3 Usage Scenario

MAP is applicable in scenarios where method names need to be updated. MAP can assist developers in JIT method name updating. Before developers commit changes locally after modifying the method body, method name update suggestions for developers could be automatically generated by using a plugin that encapsulates the MAP method. Even if the suggested method names provided by MAP are only partially correct, they can reduce the amount of editing required by developers and improve development efficiency.

8. Conclusion and Future Work

We first propose the task of JIT Method Name Updating for reducing and avoiding inconsistent method names, while also first constructing a dataset for this specific task. In this paper, an empirical study is performed to investigate the relationship between method body modifications and method names, and we propose a combined approach based on heuristic rules and a neural model. It can generate recommended new method names based on the old method name and the modified method bodies. By conducting experiments on the built data with over 108K method name-body co-change samples, the results demonstrate that our method outperforms three baselines across all evaluation metrics, indicating the effectiveness of our work in the task of JIT Method Name Updating.

In the future, we plan to further investigate the editing patterns of method name updates and conduct research in different programming languages. Additionally, we intend to explore the use of more advanced techniques, such as pre-trained models, to address the limitations of the current methods.

Acknowledgment

This work was supported in part by the National Key Research and Development Project (No. 2021YFB1714200), the National Natural Science Foundation of China (No. 62372071), the Fundamental Research Funds for the CentralUniversities (No. 2022CDJDX-005), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2022TIAD-KPX0067), and the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538).

REFERENCES

[1] Difflib — Helpers for computing deltas. https://docs.python.org/3/library/difflib.html.

[2] GitHub - c2nes/javalang: Pure Python Java parser and tools. https://github.com/c2nes/javalang.

[3] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc. Can Lexicon Bad Smells Improve Fault Prediction? In *2012 19th Working Conference on Reverse Engineering*, pages 235–244, Oct. 2012.

[4] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus. The Effect of Lexicon Bad Smells on Concept Location in Source Code. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pages 125–134, Sept. 2011.

[5] M. Allamanis, ET. Barr, C. Bird, C. Sutton, SC. Cheung, A. Orso, and MA. Storey. Learning Natural Coding Conventions. In *22ND ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE 2014)*, pages 281–293, 2014.

[6] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to Represent Programs with Graphs, May 2018.

[7] U. Alon, S. Brody, O. Levy, and E. Yahav. Code2seq: Generating Sequences from Structured Representations of Code. *arXiv:1808.01400 [cs, stat]*, Feb. 2019.

[8] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. Code2vec: Learning distributed representations of code. *code2seq: Generating Sequences from Structured Representations of Code*, 3(POPL):40:1–40:29, Jan. 2019.

[9] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2018.

[10] V. Arnaoudova, M. Di Penta, and G. Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, Feb. 2016.

[11] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc. REPENT: Analyzing the Nature of Identifier Renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, May 2014.

[12] 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*, 40(7):671–694, July 2014.

[13] I. I. Brudaru and A. Zeller. What is the long-term impact of changes? In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, RSSE '08, pages 30–32, Nov. 2008.

[14] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In *2009 16th Working Conference on Reverse Engineering*, pages 31–35, 2009.

[15] S. Butler, M. Wermelinger, YJ. Yu, H. Sharp, and IEEE Comp Soc. Exploring the Influence of Identifier Names on Code Quality: An empirical study. In *14TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR 2010)*, pages 156–165, 2010.

[16] S. Butler, M. Wermelinger, YJ. Yu, H. Sharp, and M. Mezini. Improving the Tokenisation of Identifier Names. In *ECOOP 2011 - OBJECT-ORIENTED PROGRAMMING*, volume 6813, pages 130–154, 2011.

[17] S. Fakhoury, YZ. Ma, V. Arnaoudova, O. Adesope, and IEEE Comp Soc. The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. In *2018 IEEE/ACM 26TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC 2018)*, pages 286–296, 2018.

[18] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 313–324, 2014.

[19] 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*, pages 1034–1036, 2011.

[20] E. Grave, P. Bojanowski, P. Gupta, A. Joulin, and T. Mikolov. Learning Word Vectors for 157 Languages. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, May 2018.

[21] L. Guerrouj, Z. Kermansaravi, V. Arnaoudova, BCM. Fung, F. Khomh, G. Antoniol, and YG. Gueheneuc. Investigating the relation between lexical smells and change- and fault-proneness: An empirical study. *SOFTWARE QUALITY JOURNAL*, 25(3):641–670, Sept. 2017.

[22] T. Hoang, H. J. Kang, J. Lawall, and D. Lo. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, June 2020.

[23] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi. DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45, May 2019.

[24] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[25] J. Hofmeister, J. Siegmund, and D. V. Holt. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 217–227, 2017.

[26] E. W. Høst and B. M. Østvold. Debugging Method Names. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, volume 5653, pages 294–317. 2009.

[27] S. Kim and D. Kim. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering*, 21(2):565–604, Apr. 2016.

[28] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Whats in a Name? A Study of Identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12, 2006.

[29] V. I. Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[30] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated Graph Sequence Neural Networks, Sept. 2017.

[31] Y. Li, S. Wang, and T. Nguyen. A context-based automated approach for method name consistency checking and suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 574–586, 2021.

[32] B. Liblit, A. Begel, and E. Sweetser. Cognitive perspectives on the role of naming in computer programs. In *PPIG*, page 11, 2006.

[33] B. Lin, S. Wang, K. Liu, X. Mao, and T. F. Bissyande. Automated Comment Update: How Far are We? In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 36–46, May 2021.

[34] B. Lin, S. Wang, Z. Liu, X. Xia, and X. Mao. Predictive Comment Updating with Heuristics and AST-Path-Based Neural Learning: A Two-Phase Approach. *IEEE Transactions on Software Engineering*, pages 1–20, 2022.

[35] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin. Learning to recommend method names with global context. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1294–1306, May 2022.

[36] 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 41st International Conference on Software Engineering*, ICSE '19, pages 1–12, May 2019.

[37] Z. Liu, X. Xia, D. Lo, M. Yan, and S. Li. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering*, 2021.

[38] Z. Liu, X. Xia, M. Yan, and S. Li. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 585–597, Dec. 2020.

[39] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen. Suggesting natural method names to check name consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1372–1384, June 2020.

[40] M. Pradel and K. Sen. DeepBugs: A Learning Approach to Name-Based Bug Detection. *PROCEEDINGS OF THE ACM ON PROGRAMMING LANGUAGES-PACMPL*, 2, Nov. 2018.

[41] Z. Qu, Y. Hu, J. Zeng, B. Cai, and S. Yang. Method Name Generation Based on Code Structure Guidance. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1101–1110, Mar. 2022.

[42] A. Schankin, A. Berger, DV. Holt, JC. Hofmeister, T. Riedel, M. Beigl, and IEEE Comp Soc. Descriptive Compound Identifier Names Improve Source Code Comprehension. In *2018 IEEE/ACM 26TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC 2018)*, pages 31–40, 2018.

[43] A. See, PJ. Liu, and CD. Manning. Get To The Point: Summarization with Pointer-Generator Networks. In R. Barzilay and MY. Kan, editors, *PROCEEDINGS OF THE 55TH ANNUAL MEETING OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS (ACL 2017), VOL 1*, pages 1073–1083, 2017.

[44] 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.*, 4(3):143–167, 1996.

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[46] S. Wang, M. Wen, B. Lin, and X. Mao. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 741–753, Aug. 2021.

[47] M. Wen, RX. Wu, and SC. Cheung. Locus: Locating Bugs from Software Changes. In D. Lo, S. Apel, and S. Khurshid, editors, *2016 31ST IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE)*, pages 262–273, 2016.

[48] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep Learning Code Fragments for Code Clone Detection. In D. Lo, S. Apel, and S. Khurshid, editors, *2016 31ST IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE)*, pages 87–98, 2016.

[49] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt. Learning to represent edits. *arXiv preprint arXiv:1810.13337*, 2018.

[50] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang. Deep just-in-time defect prediction: How far are we? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 427–438, 2021.