Automating Method Naming with Context-Aware Prompt-Tuning

Jie Zhu[†‡], Lingwei Li[†‡], Li Yang[*†], Xiaoxiao Ma[†], Chun Zuo[§]
[†]Institute of Software, Chinese Academy of Sciences, Beijing, China
[‡]University of Chinese Academy of Sciences, Beijing, China
[§]Sinosoft Co.,Ltd., Beijing, China
{zhujie212, lilingwei20}@mails.ucas.edu.cn
{yangli2017, xiaoxiao}@iscas.ac.cn
zuochun@sinosoft.com.cn

***Abstract*—Method names are crucial to program comprehension and maintenance. Recently, many approaches have been proposed to automatically recommend method names and detect inconsistent names. Despite promising, their results are still sub-optimal considering the three following drawbacks: 1) These models are mostly trained from scratch, learning two different objectives simultaneously. The misalignment between two objectives will negatively affect training efficiency and model performance. 2) The enclosing class context is not fully exploited, making it difficult to learn the abstract functionality of the method. 3) Current method name consistency checking methods follow a generate-then-compare process, which restricts the accuracy as they highly rely on the quality of generated names and face difficulty measuring the semantic consistency.**

In this paper, we propose an approach named AUMENA to AUtomate MEthod NAming tasks with context-aware prompt-tuning. Unlike existing deep learning based approaches, our model first learns the contextualized representation(i.e., class attributes) of programming language and natural language through the pre-training model, then fully exploits the capacity and knowledge of large language model with prompt-tuning to precisely detect inconsistent method names and recommend more accurate names. To better identify semantically consistent names, we model the method name consistency checking task as a two-class classification problem, avoiding the limitation of previous generate-then-compare consistency checking approaches. Experiment results reflect that AUMENA scores 68.6%, 72.0%, 73.6%, 84.7% on four datasets of method name recommendation, surpassing the state-of-the-art baseline by 8.5%, 18.4%, 11.0%, 12.0%, respectively. And our approach scores 80.8% accuracy on method name consistency checking, reaching an 5.5% outperformance. All data and trained models are publicly available.

***Index Terms*—Method Name Recommendation, Inconsistent Method Name Checking, Prompt Tuning.**

I. Introduction

Program comprehension is the foundation of software evolution and maintenance [1]. Developers need to understand programs written by themselves or others before making any modifications. Among all the factors affecting the program's understandability, method names play a significant role as they are brief summaries of source code and could indicate the developer's purpose [2]. Recent studies show that some programmers even take notes of crucial method names to help them figure out the application procedure, which further demonstrates its significance in program comprehension [3].

However, method names could al so be confusing, making programs even harder to understand [4, 5] and more error-prone [6]. To improve the readability and maintainability of software, developers need to rename methods when their original names are of poor quality or their implementation logic has been changed [7]. Nevertheless, naming is a time-consuming and non-trivial task for developers [8]. Poor method naming still widely occurs in many projects for a variety of reasons including insufficient communication and lack of knowledge about the project [9, 10, 11, 12]. Besides, according to a recent study, developers often discussed renaming in pull request reviewing activities, aiming at a large range of objectives from fixing typos to keeping naming consistency and better reflecting code responsibility [13]. Therefore, automatically detecting inconsistent method names and recommending better method names are especially practical for real-world projects and could boost the researches of related areas such as Code Review [14, 15], Code Smell [16], and Code Refactoring [7, 13].

Recently, various automatic approaches have been proposed on the two tasks of method naming: 1) Method name Consistency Checking(MCC), and 2) Method Name Recommendation(MNR). With the idea that similar methods should have similar names, early studies [8, 17, 18] such as Code2Vec [18] and Mercem [17] mainly employ information retrieval(IR) techniques to recommend the names of similarly implemented methods. However, these IR-based methods fail to generate new names which have never been seen before. Code2Seq [19] further improves IR-based Code2Vec [18] by importing seq2seq models to recommend method names. Nguyen et al. [20] proposed MNire which utilized the enclosing class name to enhance method name suggestion. Li et al. [21] developed DeepName and extended the model input context to surrounding and interacting methods. Wang et al. [12] and Liu et al. [22] further analyzed and combined contexts from different levels, achieving state-of-the-art results on MNR. For the task of method name consistency checking, the state-of-the-art approach Cognac [12] follows the same strategy of MNire [20], which checks method name consistency by

[*]Li Yang is the corresponding author.

generating a method name first and then computing the lexical similarity between the newly generated name and current name. If the calculated similarity is lower than the selected threshold, the current name will be labeled as inconsistent.

Despite promising results, these previous method naming approaches have three principal limitations. **1) All models of deep learning based methods are trained from scratch, learning two distinct objectives simultaneously**: one is to learn the semantic representation of programming language and natural language, and another is about learning the relationship between the method name and its implementation to check consistency and recommend better names. The misalignment between the two optimizing objectives decreases the efficiency of training and thus leads to sub-optimal results. **2) Most recent method name consistency checking approaches, including MNire [20] and Cognac [12], follow a generate-then-compare strategy to detect inconsistent method names, facing difficulty measuring the semantic consistency.** These MCC models determine whether a method name is consistent by calculating the lexical similarity between the current method name and newly generated name. If the calculated lexical similarity is lower than the selected threshold, the original method name will be labeled as inconsistent, otherwise not. However, method names with completely different sub-tokens could be semantically similar while names with low lexical similarity could have totally different meanings. For example, method names "*create_form_data*" and "*delete_form_data*" are similar in literal, but convey the opposite meanings. And previous generate-then-compare MCC methods will incorrectly predict this example to be consistent as they only consider the lexical similarity. **3) The context in the enclosing class is not fully exploited.** Previous studies [12, 20, 21] focus on the class name and the sibling methods without considering class attributes.

To mitigate the above issues, we propose AUMENA, a method naming automation approach. First, we adopt the "pre-train, prompt, and predict" paradigm to detect inconsistent method names and generate high-quality names. Specifically, with the pre-trained model that has learned the optimal contextualized representation of code tokens and natural texts in advance, AUMENA could concentrate more on the downstream tasks of method naming. Besides, the CodeT5 [23] pre-trained model adopted in our approach has been specially pre-trained using the identifier tagging and masked identifier prediction tasks. These two elaborately designed tasks highlight the importance of identifiers in source code, which perfectly fit our model and tasks since identifiers play a vital role in method naming. In addition, we involve prompt tuning to bridge the gap between pre-training and tuning on downstream tasks, which contributes to fully exploiting the knowledge and capacity of the pre-trained model. Second, AUMENA models the method name consistency checking task as a 2-class classification problem. Given the method name and the method contexts, the prompt-based binary classification model will directly predict whether the method name is consistent, avoiding the disadvantages of calculating lexical similarity. The major challenge in building the classification model lies in how to collect and construct sufficient inconsistent method names for training, as they should be related but not consistent with the implementation. To cope with this challenge, we propose a novel hard negative sampling method to generate sufficient high-quality negative training samples for the classification model. Finally, AUMENA involves class attributes to enrich the enclosing context, which proves to be effective in method name recommendation.

To evaluate the effectiveness of AUMENA on method name recommendation task, we conducted experiments following Cognac [12] and GTNM [22] on four widely-adopted datasets, including *Java-small*, *Java-med* and *Java-large* from Alon et al. [19] and another one built by Nguyen et al. [20]. Experimental results reflect that AUMENA outperforms all state-of-the-art approaches by a large margin (e.g., AUMENA surpass existing techniques by 8.5%, 18.4%, 11.0%, and 12.0%, respectively, on F-score of four datasets). For the method name consistency checking task, our approach also achieves better performance on the test set collected by Liu et al. [8], improving from 76.6% to 80.8% on the overall accuracy compared with the state-of-the-art Cognac [12].

The main contributions of our approach are outlined as follows:

- To the best of our knowledge, we are the first to leverage prompt-tuning on method naming, which exploits the potential of pre-trained models by filling the gap between the pre-training tasks and downstream naming tasks.
- We propose a prompt-based binary classification approach to detect inconsistent names, which is capable of measuring the semantic consistency between method name and its implementation.
- Experiment results on five widely-used datasets show that AUMENA performs better than all state-of-the-art approaches by a large margin on both the method name recommendation and method name consistency checking tasks. And the quality of AUMENA-generated method names is similar or even higher than human-written ones.

All trained models and data are publicly available [1].

II. Motivating Examples

Figure 1 shows three motivating examples in our dataset. Example 1 and 2 are on the method name recommendation task and Example 3 is about detecting inconsistent names.

In Example 1, it shows that existing models suffer from a discrepancy in that method names sometimes could not be extracted from tokens of program entities directly. It can be observed that the method aims to change the name of the object "value" with "bind" and "unbind" operations. However, "unbind" recomended by existing MNR model is just one of the two operations to activate the "oldName" variable. And the whole statement of the two operations "bind(newName, value); unbind(oldName);" is too long to be a method name. e.

[1]https://figshare.com/s/0382ba979d970b4c2b23

Index	Existing Model	Ground Truth	Analysis

1	public void unbind(Name oldName, Name newName) throws NamingException
	{
	  Object value = lookup(oldName);
	  bind(newName, value);
	  unbind(oldName);
	}
		public void rename(Name oldName, Name newName) throws NamingException
		{
		  Object value = lookup(oldName);
		  bind(newName, value);
		  unbind(oldName);
		}
			Method name "rename" is a high-level summary of the method's abstract functionality, which demands higher on understanding the semantics of the key statements.

2	public final class ComponentOrientation implements java.io.Serializable
	{
	  ...
	  public static final ComponentOrientation LEFT_TO_RIGHT =
	        new ComponentOrientation(HORIZ_BIT|LTR_BIT);
	  ...
	  public boolean isVertical() {
	    return (orientation & LTR_BIT) != 0;
	  }
	}
		public final class ComponentOrientation implements java.io.Serializable
		{
		  ...
		  public static final ComponentOrientation LEFT_TO_RIGHT =
		        new ComponentOrientation(HORIZ_BIT|LTR_BIT);
		  ...
		  public boolean isLeftToRight() {
		    return (orientation & LTR_BIT) != 0;
		  }
		}
			Class Attribute "LEFT_TO_RIGHT" serves as an essential hint for the model to recommend method name correctly.

3	protected void doBindCurrentConfiguration(Context context) {
	  super.bindCurrentConfiguration(context.getConfiguration());
	}
		# Actually, the method name "doSetup" is consistent with its body.
		protected void doSetup(Context context) {
		  super.bindCurrentConfiguration(context.getConfiguration());
		}
			The two method names are semantically similar but have low lexicial similarity. As a result, the MCC model might misclassify the original name "deSetup" as inconsistent.

Fig. 1. Motivating Examples

In that case, we intend to improve it by leveraging pre-trained models since they are able to understand the semantics of the key statement and summarize it as a phrase ”rename”.

Example 2 manifests the significance of introducing class attributes in Method Name Recommendation. From the example, we can learn that the input method is not sufficient to present the entire function of the code. And that's why the prior study failed to recommend this name. However, we observed that the context of class attributes could help to fill up the lack. In this situation, the statement above ”public static final ComponentOrientation LEFT_TO_RIGHT = new ComponentOrientation(HORIZ_BIT|LTR_BIT);” provides a target variable the method will work on. We improve the framework of method name recommendation following this message.

Example 3 reflects the limitation of previous generate-then-compare method name consistency checking approaches. They detect inconsistent names by calculating the lexical similarity between the original method name and generated name. According to this case, the original method name ”doSetup” is consistent with its body. However, existing consistency checking model marks it as inconsistent in that the lexical similarity between ”doSetup” and the recommended name ”doBindCurrentConfiguration” is relatively small, even though they are semantically similar. This further demonstrates the disadvantages of previous generate-then-compare methods. And we could develop a classification-based consistency checking approach to avoid the drawback.

III. Related Work

A. Method Name Recommendation

Recently, many automatic approaches have been proposed to ease the burden of method naming. By constructing two vector spaces of names and bodies to compute similarities, Liu et al. [8] built a deep learning model to recommend method names by retrieving names from most similar methods. Yonai et al. [17] proposed Mercem, which suggests method names based on the method embedding obtained from the caller-callee relationship. Code2Vec [18] exploits structural properties within a code snippet by introducing the abstract syntax tree (AST) and then retrieves the most semantically similar method names. However, the IR-based methods above face difficulty generating new names have never seen before, making them harder to be applied in practice. To tackle the above issue, Code2Seq [19] further improves Code2Vec [18] by using the seq2seq model to generate method names. Allamanis et al. [24] developed a neural probabilistic language model for learning the embedding of source code tokens to generate method names. Later they improved their model by utilizing a neural convolutional attentional network, which requires no hard-coded features to extract local time-invariant and long-range topical attention features [25]. Xu et al. [26] used the hierarchical neural network to suggest method names with block structures. The limitation of these works above is that only the context within the target itself is leveraged. To overcome the restriction, Nguyen et al. [20] proposed MNire which utilized program entities including the enclosing class name to recommend method names. Following the idea of MNire, Li et al. [21] developed DeepName, which extends the model input context to surrounding and interacting methods, such as caller, callee, and sibling methods. Wang et al. [12] conducted an empirical study on the relationship between method name and diverse specific contexts and proposed Cognac, a method name recommendation model based on pointer-generator network [27]. Ge et al. [28] built a two-phase keywords guided approach which decomposes the method naming task into keywords extraction and method name generation tasks. Qu et al. [29] proposed SGMNG, a structure-guided method name recommendation approach combining semantic and structural features with a graph neural network. Liu et al. [22] built a Transformer-based neural model for method name suggestion, considering contexts of different levels simultaneously.

B. Method Name Consistency Checking

The method name consistency checking task is to determine whether a given method name is consistent with its implementation, which could also be viewed as the first step prior to method name recommendation. Høst and Østvold [30] studied the method naming conventions and utilized the convention to extract static rules to debug method names. Kim et al. [11] relied on the code dictionary built from API documents to detect inconsistent identifiers including method names. With the intuition that the methods with similar implementation should have similar names, Liu et al. [8] detected inconsistent method names by calculating the similarity between the set of names with similar method name embedding and the set of names with similar method implementation embedding. However, methods sharing similar implementations could still have different names as they might exist in different classes and projects [22]. To overcome this shortcoming, in recent years, MNire [20] and DeepName [21] utilized multiple contexts to enhance method name consistency checking. The state-of-the-art approach Cognac [12] further explored different contexts and followed the same strategy of MNire [20], which determines whether a method name is consistent by calculating the lexical similarity between the given method name and the name generated by the method name recommendation model.

C. Prompt Tuning

In recent years, pre-trained models for source code (e.g., CodeT5 [23], CodeBERT [31], GraphCodeBERT [32]) have achieved remarkable success in various code understanding and generation tasks. Most existing researches follow a general "pre-train, fine-tune" paradigm. Specifically, the pre-trained models are first pre-trained on a large corpus of source code using the Masked Language Modeling (MLM) objective and then fine-tuned on a smaller dataset to adapt themselves to the downstream tasks. However, the inputs and optimizing objectives differ during the pre-training and fine-tuning phases. This misalignment makes the knowledge and capacity of pre-trained models hard to be fully exploited and thus limits their performance on the downstream tasks.

To fill the gap between pre-training and fine-tuning, there are two main research directions: one direction is to redesign the pre-training tasks to make them more similar to the downstream tasks [33], and another is to reformulate the downstream tasks into a fill-in-blank form with prompt which resembles the MLM task in pre-training [34]. Prompt-tuning follows the second direction to mitigate the issues of fine-tuning by providing a prompt template such as "The method name is [*MASK*]" at the end of the input. The training objective of prompt-tuning is to predict the masked token [*MASK*]. The model could further decide whether the method name is consistent using a verbalizer which maps the predicted word to the class. And this goes the same for method name recommendation. Through the improvement of converting the downstream task into the fill-in-blank task, the pre-trained models conduct the downstream task in the same way during pre-training, which assists in bridging the gap and consequently boosts the performance.

IV. Approach

AUMENA mainly focuses on two different but related method naming tasks with prompt-tuning: 1) Method Name Consistency Checking (MCC) and 2) Method Name Recommendation (MNR). We model the MCC task as a 2-class classification problem and the MNR task as a bi-model summarization problem. The overall architecture of our prompt-tuning based method naming approach is depicted in Figure 2. We adopt a novel "pre-train, prompt, and predict" paradigm to detect inconsistent method names and generate high-quality candidates. Specifically, for the method name consistency checking task, we propose a novel hard negative sampling method to generate sufficient related but inconsistent method names for training. Afterwards, with the training data collected from the corpus and the sampling approach, we train a 2-class classification model with prompt-tuning to detect inconsistency between method names and their implementations. For the method name recommendation task, AUMENA extracts different contexts from the target method and integrates them with the prompt template we defined as a sequence to feed into the prompt-tuned CodeT5. Compared with the existing approaches based on deep learning, our CodeT5-based model could concentrate more on the task of method naming with the identifier-aware pre-training. Besides, prompt-tuning also helps the pre-trained model better adapt to the two downstream naming tasks by reformulating them into cloze-style fill-in-blank problems that are closer to the pre-training tasks, thus achieving better performance.

A. Data Preparation

Following the settings of most recent studies [12, 22], we extract method contexts from different levels and divide the names of these program entities into sequences of sub-tokens using javalang [2] and spiral [3]. Specifically, the contexts employed by our approach mainly consist of two parts: (1) local implementation, such as the identifier names and method signature; (2) enclosing class, including class name, siblings, and class attributes. The contexts from identifier names and enclosing classes could provide information on the method's function, while the contexts from siblings and other contextual method names help the model adapt to the naming conventions within specific projects. After extracting the necessary contexts, we filter all duplicates to avoid data inflation and remove empty methods. Finally, we convert all input contexts into lowercase sub-token sequences using spiral, as the identifier names are usually compound words.

Considering the length of the sub-token sequence of each context might vary among different examples, we normalize the input data by restricting the length of each context. For example, some classes might have lots of attributes and methods, which might significantly increase the length of these contexts, thus affecting the overall performance as the model might ignore other shorter contexts. Consequently, we limit the number of sibling methods and class attributes to ten. And the maximum length of total input sub-token sequences is set to 512.

[2]https://github.com/c2nes/javalang
[3]https://github.com/casics/spiral

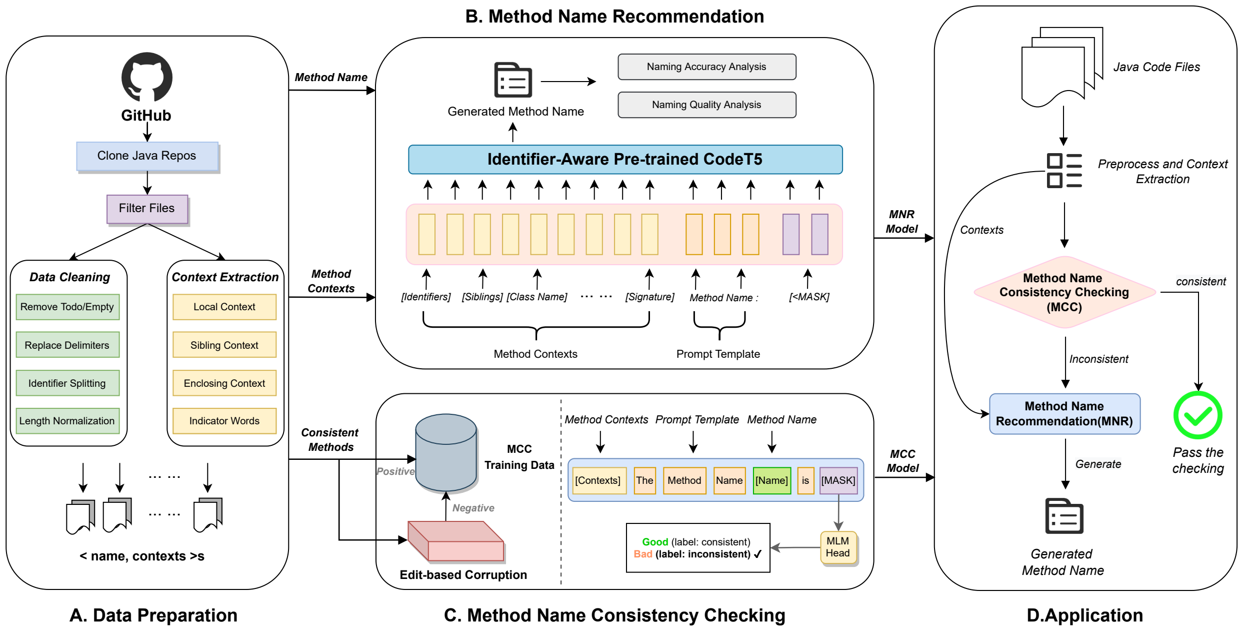


Fig. 2. Overview of AUMENA

B. Method Name Recommendation

In method name recommendation, we employ similar contexts with method name consistency, except that we further involve class contexts as they are not available in the MCC dataset of Liu [8]. Compared to previous studies on MNR, we ease the model's burden of learning the representation of programming and natural language by prompt-tuning the identifier-aware pre-trained model CodeT5 [23], which could provide a better initialization and help the model concentrate more on learning the relationship between method names and their contexts. Specifically, AUMENA will concatenate all the method contexts with indicator words and then append a natural language prompt template text to guide the model to output method names. For evaluation, we utilize an automated method name quality accessing tool [35] in addition to the quantitative metrics on accuracy.

To train the method name recommendation model, we use a cross-entropy loss here to measure the difference between the prediction and the truth. Equation 1 accumulates the sum of each sentence predicting errors (i.e., the predicted words $\hat{y}_{ij}^D$ and the manual ones y_{ij}^D in dataset D). And the model is trained to reduce the loss as follows.

$$LOSS_{MNR} = -\frac{1}{|D|}\sum_{i=1}^{|D|}\sum_{j=1}^{|Y|} y_{ij}^D log(\hat{y}_{ij}^D) \quad (1)$$

C. Method Name Consistency Checking

1) Background: Similar to prior work [8, 12, 20] on Method name Consistency Checking(MCC), AUMENA takes a method with its name and contexts as input and predicts whether the given method name is consistent with its implementation. However, previous state-of-the-art approaches [12, 20] of method name consistency checking follow a generate-then-compare strategy. They detect inconsistent method names by generating a new method name given its implementation first and then calculating the lexical similarity between the generated name and the original name. If the similarity calculated is lower than the selected threshold, the original method name will be regarded as inconsistent. However, the limitation of these approaches is that the lexical similarity could not reflect the semantics of words. Two method names with totally different words could still have similar meanings. To address the above issues, we propose to model this task as a binary classification problem. The prompt-based 2-class classification model takes the method name with its implementation together and directly predicts whether the method name is consistent. And the main challenge of building this classification model lies in collecting sufficient training data, especially the inconsistent examples.

2) Hard Negative Sampling to Build MCC Training Data: To acquire adequate training data for classification, we regard the methods from the datasets of method name recommendation as consistent examples, as they have been stable for a long time [21]. Nevertheless, inconsistent methods are somewhat difficult to collect, and there are no large-scale open datasets currently. To the best of our knowledge, the only inconsistent naming dataset is collected by Liu [8]. However, it only contains 2805 inconsistent examples, which are totally inadequate for training. To cope with this challenge, we propose a novel hard negative sampling method to build sufficient inconsistent naming examples from consistent ones. And the main difficulty here is about how to generate related but inconsistent names. For example, if we randomly choose names from other methods in the corpus as inconsistent examples, the classification model could easily identify them as they are totally unrelated to the method implementation. However, inconsistent method names are usually closely related to their contexts, and most of them could be transformed into consistent ones by several edits. Motivated by this observation, we decided to take an edit-based method to generate fake inconsistent names from consistent ones. Specifically, for a given consistent method name, we randomly corrupt the sub-tokens of names with the probability estimated from Liu's datasets, and the operations conducted on each token include "remain the same", "add", "delete" and "replace". Finally, we generate 4M inconsistent names and combine them with positive examples to form the training dataset of the method name consistency checking model.

3) Prompt-based 2-class Classification Model: After acquiring adequate training data, we combine the method name and contexts with the prompt template to optimize our classification model. Given a training example x with its name n and contexts c, the prompt-based classification model will predict the probability distribution among the label words. And a verbalizer v will map the predicted label word to the corresponding class, which is the final output of the classification model. And the loss of MCC is as follows:

$$LOSS_{MCC} = -\frac{1}{|D|}\sum_{i=1}^{|D|}\delta_i^D log(\hat{\delta}_i^D) \tag{2}$$

A cross-entropy loss is employed to measure the deviation between the prediction and the truth. Equation 2 accumulates the sum of each true-or-false classification entropy (i.e., the predicted $\hat{\delta}_i^D$ and the golden one δ_i^D in dataset D).

D. Application

After finishing the training of the MNR and MCC models, AUMENA automates the method naming tasks in three steps. First, we extract the necessary contexts, including identifiers, signature, and siblings, etc., from given java source files and send them into the MCC model. In the second step, the MCC model will determine whether the given method name is consistent with its implementation with prompt-based 2-class classification. If the predicted output is positive(consistent), the process will end as the given method passes the consistency check. If negative, the MNR model will take the contexts extracted from the first step to generate new candidate names. And we call this process consisted of three steps as the method naming automation problem.

V. Experimental Design

A. Research Questions

To evaluate our proposed AUMENA approach, we conduct experiments to answer the following research questions.

RQ1: How does AUMENA perform on the method name recommendation task against other state-of-the-art methods?

RQ2: How does AUMENA perform on the method name consistency checking task against other baselines?

RQ3: How effective is the design of modeling method name consistency checking task as a prompt-based binary classification problem?

RQ4: How is the quality of method names generated by AUMENA, compared with human-written ones?

B. Datasets

For method name recommendation tasks, we used four different datasets for evaluation following Cognac [12] and GTNM [22]. As shown in Table I, three datasets, including *Java-small*, *Java-med*, and *Java-large* from Alon et al. [19], contain 719K, 3.47M and 14.2M Java method examples, respectively. And another dataset built by Nguyen et al. [20] has 20.8M Java examples. The first three datasets provide concrete method data with the partition of training, validation, and test set, which could be downloaded directly. For the dataset of MNire by Nguyen et al. [20], we follow the same settings and partition of GTNM [22] to preprocess and build the dataset. To avoid data leakage, all method examples are split by projects instead of file-based validation, as there might be similar methods in the same project from which models could learn the coding convention of test data [12, 36].

For method name consistency checking tasks, there is no available large training dataset due to the lack of negative samples. Therefore, the dataset we used for our classification model is built from the method name recommendation corpus of GTNM [22] using the hard negative sampling approach described in IV-C. To compare our approach against other methods, we evaluated AUMENA on the test set collected by Liu et al. [8], which has been widely adopted for evaluation by previous studies [12, 20, 21].

C. Metrics

For method name recommendation, we followed prior works and used traditional Precision, Recall, and F1-score scores to evaluate our model and other baselines. These metrics measure the similarity between the prediction and the ground truth at the word level. Precision calculates the rate of matches to total predictions. Recall is the rate of matches to all samples in truth. And F1-score is the harmonic mean of Precision and Recall. To better evaluate the generation of the entire names, we also employed Exact Match Accuracy to take the order of sub-tokens into consideration, which proves to be more precise according to recent studies [12, 22].

For method name consistency checking, we used Precision, Recall, and F1-score for classification, too. In addition, Overall Accuracy was employed to measure how many generations, both positive and negative, were correctly classified, referring to golden ones.

D. Experiment Settings

1) Method Name Recommendation. In this part, we leveraged CodeT5 [23] and followed the initialization of its pre-training work. For prompt-tuning, we trained our model in 1,036K steps and evaluated at every 100K step. Some key hyperparameters are shown in Table II.

TABLE I
STATISTICS OF THE MNR DATASETS

Datasets	**Train**	**Validation**	**Test**
Java-small	643K	31K	45K
Java-median	2,711K	389K	369K
Java-large	13,442K	305K	403K
MNire's	16,580K	3,982K	267K

TABLE II
TRAINING HYPERPARAMETERS

	Hyperparameter	Value
Method Name Recommendation	Learning rate	5e-5
	Max input length	512
	Max output length	16
	Beam size	10
	Batch size	16
Method Name Consistency Checking	Learning rate	5e-5
	Max input length	512
	Batch size	16

TABLE III
RESULTS OF METHOD NAME RECOMMENDATION

Dataset	Approach	Precision	Recall	F1-score	EM Acc
Java-small	Code2vec	23.4	22	21.4	-
	MNire	44.8	38.7	41.5	15.5
	Cognac	67.1	59.7	63.2	-
	AUMENA	**69.6**	**67.6**	**68.6**	**44.3**
Java-med	Code2vec	36.4	27.9	31.9	-
	MNire	62.0	57.6	59.7	36.2
	Cognac	64.8	57.3	60.8	-
	AUMENA	**72.6**	**71.4**	**72.0**	**50.9**
Java-large	Code2vec	44.2	38.3	41.6	-
	MNire	63.1	59.0	61	37.4
	Cognac	71.4	61.9	66.3	-
	AUMENA	**74.0**	**73.2**	**73.6**	**55.3**
MNire's	Code2vec	51.9	39.8	45.1	35.6
	MNire	66.3	62.1	64.2	42.6
	Cognac	70.2	66.8	68.5	-
	GTNM	77.0	74.1	75.6	62.0
	AUMENA	**85.1**	**84.3**	**84.7**	**71.0**

2) Method Name Consistency Checking. Similar to Method Name Recommendation, we selected pretrained CodeT5 model to implement consistency checking, and it took 520K steps for the entire prompt-tuning and was evaluated at every 50K step. Also, Table II presents some key hyperparameters we set in experiments.

3) Experiment Environment. We implemented all training with three NVIDIA GeForce RTX 3090Ti GPU and CUDA 11.7[4]. To sum up, it took 100 hours for Method Name Recommendation and 23 hours for Method Name Consistency Checking on training, respectively.

VI. METHOD NAME RECOMMENDATION TASK

To evaluate AUMENA on method name recommendation task, we compare our approach against other baselines on four widely-adopted datasets introduced in V-B.

A. Baselines

We chose the following competing baselines to compare with AUMENA.

- **Code2vec.** Alon et al. [18] employ information retrieval(IR) techniques to recommend the names of similarly implemented methods.
- **MNire.** Nguyen et al. [20] use the sub-tokens of class name and method body to generate method names with RNN-based seq2seq model.
- **Cognac.** Wang et al. [12] explore not only the local context but also the global context to recommend method names with pointer-generator network.
- **GTNM.** Liu et al. [22] consider contexts from different levels to generate method names using a transformer-based seq2seq model.

[4]https://developer.nvidia.com/cuda-toolkit/

B. Results(RQ1)

The results in Table III show that AUMENA totally outperforms all baselines on four method name recommendation datasets. For Java-small, AUMENA achieves 69.6% on Precision, 67.6% on Recall, 68.6% on F1-score, and 44.3% on Exact Match, which outperforms baselines by at least 3.73%, 13.2%, 8.54%, respectively. And when it comes to Java-med and Java-large, the results go to over 70% on Precision, Recall, F1-score, and 50% on Exact Match. Then, we mainly focus on the performance of MNire's, the largest dataset. AUMENA scores highest on every metric and outperforms baselines by 10.5%, 13.8%, 12.0% and 14.5%. It indicates that our approach is capable of recommending method names more preciously and achieving a stable performance, whatever the situation is. Compared with pointer-generator network based Cognac [12], AUMENA could better handle long context sequences with transformer-based CodeT5 model, since RNN architecture has difficulties remembering long-term dependencies. And our proposed approach also outperformed GTNM [22] in that AUMENA could fully exploit the knowledge and capacity of pre-trained model with prompt tuning for better understanding of tokens in programming language and natural language. Thus, AUMENA could concentrate more on learning to generate method name accurately.

VII. METHOD NAME CONSISTENCY CHECKING TASK

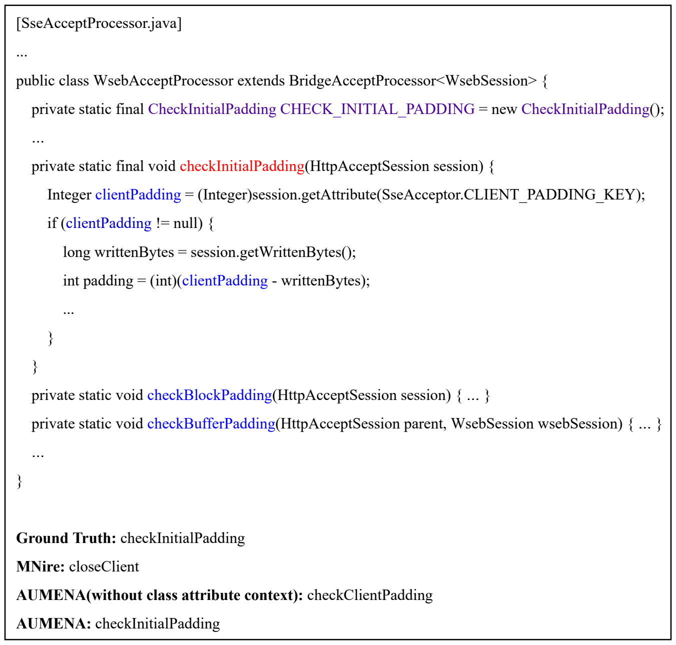
For the method name consistency checking task, we used the widely-used dataset collected by Liu et al. [8] to evaluate the effectiveness of AUMENA.

A. Baselines

We compare AUMENA with state-of-the-art baselines as follows.

TABLE IV
RESULTS OF METHOD NAME CONSISTENCY CHECKING

		DebugMethodName[8]	MNire[20]	DeepName[21]	Cognac[12]	AUMENA*	AUMENA
Inconsistent	Precision	56.8	62.7	72.3	68.6	**84.4**	81.9
	Recall	84.5	93.6	92.1	**97.6**	70.1	78.9
	F-score	67.9	75.1	**81.0**	80.6	76.6	80.4
Consistent	Precision	72.0	84.2	86.4	**96.0**	74.4	79.7
	Recall	38.2	56.0	64.8	55.6	**87.0**	82.6
	F-score	49.9	67.3	74.1	70.4	80.2	**81.1**
Overall Accuracy		60.9	68.9	75.8	76.6	78.6	**80.8**



```
[SseAcceptProcessor.java]
...
public class WsebAcceptProcessor extends BridgeAcceptProcessor<WsebSession> {
    private static final CheckInitialPadding CHECK_INITIAL_PADDING = new CheckInitialPadding();
    ...
    private static final void checkInitialPadding(HttpAcceptSession session) {
        Integer clientPadding = (Integer)session.getAttribute(SseAcceptor.CLIENT_PADDING_KEY);
        if (clientPadding != null) {
            long writtenBytes = session.getWrittenBytes();
            int padding = (int)(clientPadding - writtenBytes);
            ...
        }
    }
    private static void checkBlockPadding(HttpAcceptSession session) { ... }
    private static void checkBufferPadding(HttpAcceptSession parent, WsebSession wsebSession) { ... }
    ...
}
```

Ground Truth: checkInitialPadding
MNire: closeClient
AUMENA(without class attribute context): checkClientPadding
AUMENA: checkInitialPadding

Fig. 3. MNR Case

- **DebugMethodName.** Liu et al. [8] detect inconsistent method names by calculating the similarity between the set of names with similar method name embedding and the set of names with similar implementation embedding.
- **MNire.** Nguyen et al. [20] first generate a new method name and then compares the current name against it to detect inconsistency.
- **DeepName.** Li et al. [21] build a two-channel CNN model which takes the representation vectors of method implementation and method name to predict whether it is consistent.
- **Cognac.** Wang et al. [12] follow the same strategy of MNire [20], which computes the lexical similarity between the original name and newly generated name to check consistency.

B. Results(RQ2 & RQ3)

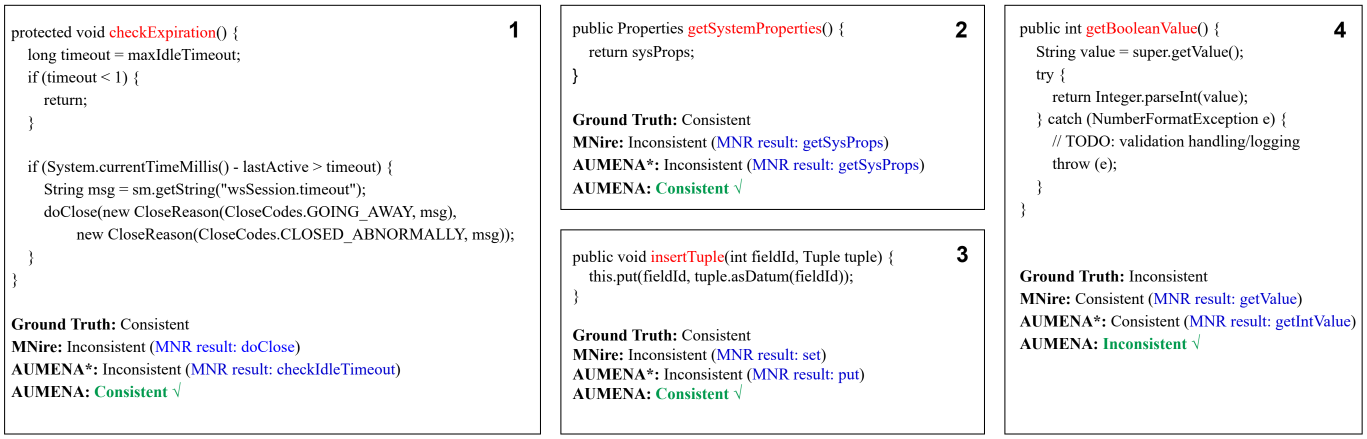
Table IV presents the results of Method name Consistency Checking(MCC). To illustrate the superiority of our classification-based approach, we also conducted experiments of AUMENA*, which takes the same strategy as MNire [20] and Cognac [12] to detect inconsistency. Specifically, in AUMENA*, we used the trained MNR model to generate a method name first and then compared the current method name against it. If the lexical similarity between them is lower than the selected threshold, AUMENA* will mark the current method name as inconsistent. In this way, we could investigate the contribution of our proposed classification-based approach by comparing results of AUMENA and AUMENA*.

It can be observed that for inconsistency checking, AUMENA achieves 80.8% on total accuracy, which leads all baselines by at least 5.5%. This proves that our model surpasses state-of-the-art on both two opposites for consistency checking, which indicates the effectiveness of AUMENA in method name consistency checking. And we also investigate the contribution of modeling MCC task as a two-class classification problem by comparing the results of AUMENA and AUMENA*. It turns out that our classification-based MCC model(AUMENA) could achieve better performance than AUMENA* on examples from both the consistent and inconsistent ones. And the overall accuracy of AUMENA increases from 78.6% to 80.8%, which further demonstrates the superiority of our prompt-based classification approach. In addition, AUMENA* achieves higher overall accuracy compared with Cognac[12] and MNire [20]. This outperformance could also support the argument that our MNR model recommends more accurate method names, since AUMENA* takes the same generate-then-compare strategy of Cognac and MNire.

VIII. DISCUSSION

A. Qualitative Analysis

1) MNR Case Study: In this section, we provide a method name recommendation example to demonstrate the context-aware ability of AUMENA. As shown in Figure 3, MNire misunderstood the method's functionality and ignored the names of input sibling methods. To further reflect the importance of class attribute context, we train two versions of AUMENA MNR models: one with context from class attribute and another without. In this case, the model with class attribute context gives the correct answer in that the crucial sub-token 'Initial' only appears in the class attribute, which serves as a key clue for the model. This illustrates that AUMENA could consider contexts from different sources to generate more accurate method names.



1

```
protected void checkExpiration() {
    long timeout = maxIdleTimeout;
    if (timeout < 1) {
        return;
    }

    if (System.currentTimeMillis() - lastActive > timeout) {
        String msg = sm.getString("wsSession.timeout");
        doClose(new CloseReason(CloseCodes.GOING_AWAY, msg),
                new CloseReason(CloseCodes.CLOSED_ABNORMALLY, msg));
    }
}
```

Ground Truth: Consistent
MNire: Inconsistent (MNR result: doClose)
AUMENA*: Inconsistent (MNR result: checkIdleTimeout)
AUMENA: **Consistent √**

2

```
public Properties getSystemProperties() {
    return sysProps;
}
```

Ground Truth: Consistent
MNire: Inconsistent (MNR result: getSysProps)
AUMENA*: Inconsistent (MNR result: getSysProps)
AUMENA: **Consistent √**

3

```
public void insertTuple(int fieldId, Tuple tuple) {
    this.put(fieldId, tuple.asDatum(fieldId));
}
```

Ground Truth: Consistent
MNire: Inconsistent (MNR result: set)
AUMENA*: Inconsistent (MNR result: put)
AUMENA: **Consistent √**

4

```
public int getBooleanValue() {
    String value = super.getValue();
    try {
        return Integer.parseInt(value);
    } catch (NumberFormatException e) {
        // TODO: validation handling/logging
        throw (e);
    }
}
```

Ground Truth: Inconsistent
MNire: Consistent (MNR result: getValue)
AUMENA*: Consistent (MNR result: getIntValue)
AUMENA: **Inconsistent √**

Fig. 4. Some examples from MCC testset

2) MCC Case Study: Figure 4 presents some examples from the testset of method name consistency checking [8]. Given a method body, MNire [20] and AUMENA* follow the same generate-then-compare strategy to detect inconsistency by calculating the similarity of newly generated name and original name. And the only difference is that MNire takes an RNN-based seq2seq MNR model, while AUMENA* utilizes our CodeT5-based MNR model introduced in IV-B to recommend method names. In contrast, AUMENA uses our proposed prompt-based classification approach to detect inconsistent method names. Different from AUMENA*, it models the MCC task as a 2-class classification problem, which allows AUMENA to measure the semantic consistency between the method name and method implementation.

In example 1, the method name ”checkExpiration” is obviously consistent with its body. However, both the MNire and AUMENA* predict it to be inconsistent, as the names suggested by their MNR models are totally different from the given name ”checkExpiration” on word-level. Even if the implication of ”checkIdleTimeout” is similar to the target name, AUMENA fails to predict correctly in that it only decides by calculating the lexical similarity between ”checkIdleTimeout” and ”checkExpiration”. And it goes the same in example 2 and 3. Actually the recommended name ”getSysProps” in example 2 conveys exactly the same meaning of ”getSystemProperties”. But as the lexical similarity between their sub-tokens is relatively small, the original method name ”getSystemProperties” is incorrectly marked as inconsistent by AUMENA* and MNire. However, AUMENA could give the correct result as it is capable of measuring the semantic consistency directly. And in example 3, generating the method name ”insertTuple” perfectly given its implementation is a challenging task. Consequently, both MNire and AUMENA* fail to give the correct MCC result as they highly rely on the accuracy of generated names. But it is much easier for AUMENA to correctly classify as it is only required to conduct a 2-class classification, instead of generating the whole method name perfectly. In example 4, the method name ”getBooleanValue” is inconsistent since the method body is about parsing the input string to an integer value instead of a boolean value. However, both MNire and AUMENA* believe that the given method name is appropriate, as their MNR results share most of the sub-tokens with the target method name. This further demonstrates the limitation of previous generate-then-compare MCC approaches in which different words could express the same meaning while method names sharing similar tokens could be totally different in semantics.

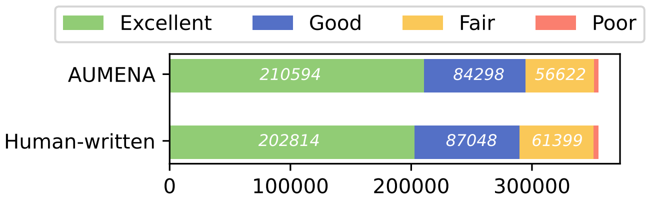


Fig. 5. Results of Naming Quality Analysis

B. Name Quality Analysis(RQ4)

Most recent method name recommendation approaches evaluate themselves with metrics of machine learning, such as precision, recall, and EM accuracy. However, method names differ from general natural language sequences because they have specific patterns and rules [37]. As a result, in the evaluation of MNR models, not only the accuracy metrics but also the quality of method names should be taken into consideration.

Specifically, we used the tool developed by Alsuhaibani et al. [35] to compare the quality of human-written method names and AUMENA recommended names. The tool implements ten verified method naming standards and produces a score from 0-10 for the given method name based on the ten standards. For example, a score of 10 means the method name follows all the standards and the name will be marked as excellent. Figure 5 shows the results of naming quality analysis on *java-large* dataset with 355k examples. It could be

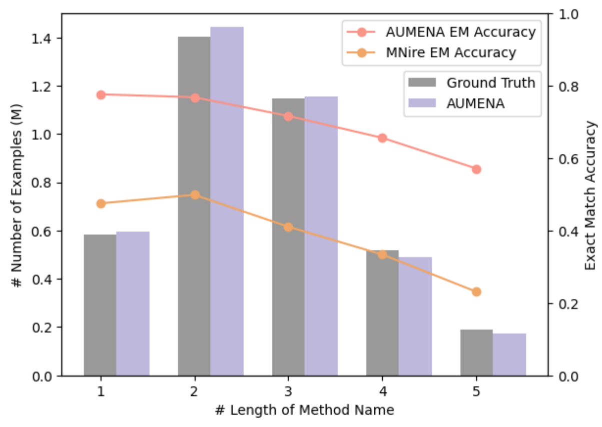


Fig. 6. The method name length distribution and the exact match accuracy with different lengths

observed that AUMENA recommends more excellent method names compared with human-written ones. And it turns out that the average score of AUMENA-generated method names is 9.23, which is even higher than the score of human-written method names 9.17. These results of quality analysis illustrate that AUMENA could generate method names with similar or even higher quality compared to human-written ones from the perspective of method naming standards.

C. Name Length Analysis

We analyze the recommended name length distribution and compare the exact match accuracy of AUMENA with MNire [20] for different name lengths in Figure 6. Specifically, we apply the Wilcoxon Signed Rank Test (WSRT) [38] to figure out whether there is a significant difference between the length distribution of AUMENA-generated method names and original names. The results reflect that there is no significant difference since all the p-values are larger than 0.05. For the exact match accuracy of method names with different lengths, we could find that the EM accuracy decreases as the MNR task with the increase of name length. The reason is that it becomes harder to predict each word correctly when the length of method name increases [22]. Still, AUMENA achieves an EM accuracy of 57.12% for method names of length 5. Compared to the the results of MNire, the EM accuracy curve of AUMENA is much flatter, which illustrates that the accuracy of method names generated by AUMENA is not only better, but also more stable on different lengths.

D. Threats to Validity

Construct validity: We followed prior studies [8, 12, 20, 21, 22] and constructed experiments on the same datasets. However, since individual knowledge varies, there is no guarantee that all method names are good enough. Also, engineers could keep their own points of view on the same naming work. Therefore, the dataset may still contain sub-optimal method names, and it can affect both the evaluations of method name recommendation and method name consistency checking tasks. Nevertheless, a handful of incorrect samples have little effect on the performance of deep learning based approaches as they learn from the majority instead of the minority [12].

Internal validity: Studies have come up with the argument that the impact of hyperparameters on DL models' effect still remains uncertain [39, 40]. To amplify the performance, we follow Tree-structured Parzen Estimator (TPE) [41] to optimize all models in this paper. However, we recognize that there may be a few more settings that could cause the same or better results. And that's also a part of our future study.

External validity: To deeply explore the insight of the tasks, we limit our approach to Java projects. Hence, the generalizability for other languages is still unseen in this paper. Future work will move on to explore the performance of AUMENA on other programming languages.

E. Application Scenario and Future Work

As the experimental results have reflected that AUMENA is more effective than other baselines on both MNR and MCC tasks, it is of significance for us to apply our approach in practice. For example, we could conduct just-in-time name recommendation in real-world software development. And it is also feasible to build a tool to detect inappropriate method names by taking a whole project as input. For the inconsistencies, we could recommend candidate names and organize them as a report for developers as reference. However, in real software engineering projects, inconsistent method names are supposed to be extremely rare. This class-imbalance problem will negatively affect the performance of MCC models in practice. Besides, to what extent the method naming automation tool helps developers remains to be explored further. In the future, we plan to develop a production-ready tool to apply AUMENA in real-world scenarios, for lightening the burden of developers on method naming.

IX. Conclusion

In this paper, we propose AUMENA, a method naming automation approach based on prompt-tuning. Unlike the prior work, AUMENA develops a prompt-based classification model to detect inconsistent method names, which is capable of measuring the semantic consistency. The novel "pre-train, prompt, and predict" paradigm we adopt helps exploit the potential of pre-trained models by filling the gap between the pre-training tasks and downstream naming tasks. The experimental results show that our approach significantly outperforms other baselines in both the method name recommendation and method name consistency checking tasks.

X. Acknowledgments

We sincerely appreciate the valuable feedback from the anonymous reviewers. This work was supported by the National Key R&D Program of China (No.2021YFC3340204), the Alliance of International Science Organizations, (Grant No. ANSO-CR-KP-2022-03), and the Strategy Priority Research Program of Chinese Academy of Sciences (No.XDA20080200).

References

[1] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *10th International Workshop on Program Comprehension (IWPC 2002), 27-29 June 2002, Paris, France*, 2002, pp. 271–278.

[2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Software Eng.*, vol. 32, no. 12, pp. 971–987, 2006.

[3] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 255–265.

[4] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. Guéhéneuc, "REPENT: analyzing the nature of identifier renamings," *IEEE Trans. Software Eng.*, vol. 40, no. 5, pp. 502–532, 2014.

[5] D. J. Lawrie, C. Morrell, H. Feild, and D. W. Binkley, "What's in a name? A study of identifiers," in *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, 2006, pp. 3–12.

[6] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, 2009, pp. 31–35.

[7] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "An empirical investigation of how and why developers rename identifiers," in *Proceedings of the 2nd International Workshop on Refactoring, IWoR@ASE 2018, Montpellier, France, September 4, 2018*, 2018, pp. 26–33.

[8] 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 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 1–12.

[9] V. Arnaoudova, M. D. Penta, G. Antoniol, and Y. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, 2013, pp. 187–196.

[10] Y. Higo and S. Kusumoto, "How often do unintended inconsistencies happen? deriving modification patterns and detecting overlooked code fragments," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, 2012, pp. 222–231.

[11] S. Kim and D. Kim, "Automatic identifier inconsistency detection using code dictionary," *Empir. Softw. Eng.*, vol. 21, no. 2, pp. 565–604, 2016.

[12] S. Wang, M. Wen, B. Lin, and X. Mao, "Lightweight global and local contexts guided method name recommendation with prior knowledge," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, 2021, pp. 741–753.

[13] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta, "Why developers refactor source code: A mining-based study," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 29:1–29:30, 2020.

[14] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *IEEE Trans. Software Eng.*, vol. 47, no. 3, pp. 595–613, 2021.

[15] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, 2022, pp. 2291–2302.

[16] X. Han, A. Tahir, P. Liang, S. Counsell, and Y. Luo, "Understanding code smell detection via code review: A study of the openstack community," in *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, 2021, pp. 323–334.

[17] H. Yonai, Y. Hayase, and H. Kitagawa, "Mercem: Method name recommendation based on call graph embedding," in *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019*, 2019, pp. 134–141.

[18] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[19] 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*, 2019.

[20] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, "Suggesting natural method names to check name consistencies," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, 2020, pp. 1372–1384.

[21] Y. Li, S. Wang, and T. N. Nguyen, "A context-based automated approach for method name consistency checking and suggestion," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, 2021, pp. 574–586.

[22] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin, "Learning to recommend method names with global context," in *44th IEEE/ACM 44th International Conference on Soft-*

ware Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, 2022, pp. 1294–1306.

[23] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, 2021, pp. 8696–8708.

[24] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 38–49.

[25] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, vol. 48, 2016, pp. 2091–2100.

[26] S. Xu, S. Zhang, W. Wang, X. Cao, C. Guo, and J. Xu, "Method name suggestion with hierarchical attention networks," in *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, 2019, pp. 10–21.

[27] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 1073–1083.

[28] F. Ge and L. Kuang, "Keywords guided method name generation," in *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, 2021, pp. 196–206.

[29] Z. Qu, Y. Hu, J. Zeng, B. Cai, and S. Yang, "Method name generation based on code structure guidance," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, 2022, pp. 1101–1110.

[30] E. W. Høst and B. M. Østvold, "Debugging method names," in *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, ser. Lecture Notes in Computer Science, vol. 5653, 2009, pp. 294–317.

[31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, vol. EMNLP 2020, 2020, pp. 1536–1547.

[32] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[33] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, and N. Sundaresan, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, 2022, pp. 1035–1047.

[34] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, 2022, pp. 382–394.

[35] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic, "An approach to automatically assess method names," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, 2022, pp. 202–213.

[36] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, 2019, pp. 602–614.

[37] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic, "On the naming of methods: A survey of professional developers," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, 2021, pp. 587–599.

[38] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics*, vol. 1, pp. 196–202, 1945.

[39] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[40] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery &amp; Data Mining*, ser. KDD '19, New York, NY, USA, 2019, p. 2623–2631.

[41] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, vol. 24, 2011.