Pre-Implementation Method Name Prediction for Object-Oriented Programming

SHANGWEN WANG, Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha, China
MING WEN, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China
BO LIN, Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha, China
YEPANG LIU, Research Institute of Trustworthy Autonoumous Systems and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg
XIAOGUANG MAO, Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha, China

Method naming is a challenging development task in object-oriented programming. In recent years, several research efforts have been undertaken to provide automated tool support for assisting developers in this task. In general, literature approaches assume the availability of method implementation to infer its name. Methods however are usually named before their implementations. In this work, we fill the gap in the literature about method name prediction by developing an approach that predicts the names of all methods to be implemented within a class. Our work considers the class name as the input: the overall intuition is that classes with semantically similar names tend to provide similar functionalities, and hence similar method names. We first conduct a large-scale empirical analysis on 258K+ classes from real-world projects to validate our hypotheses. Then, we propose a hybrid big code-driven approach, `Mario`, to predict method names based on the class name: we combine a deep learning model with heuristics summarized from code analysis. Extensive experiments on 22K+ classes yielded promising results: compared to the state-of-the-art code2seq model (which leverages method implementation data), our approach achieves comparable results in terms of F-score at token level prediction; our approach, additionally, outperforms code2seq in prediction at the name level. We further show that our approach significantly outperforms several other baselines.

† Ming Wen and Bo Lin are the corresponding authors.
This work was partially done when the first author was a visiting scholar at Southern University of Science and Technology.
Authors' addresses: Shangwen Wang, wangshangwen13@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha, China; Ming Wen, mwenaa@hust.edu.cn, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China; Bo Lin, linbo19@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha, China; Yepang Liu, liuyp1@sustech.edu.cn, Research Institute of Trustworthy Autonoumous Systems and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, Luxembourg; Xiaoguang Mao, xgmao@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha, China.

1049-331X/2023/4-ART $15.00
https://doi.org/10.1145/xxxxxxx.xxxxxxx

CCS Concepts: • **Software and its engineering → Software design techniques**; **Object oriented development**.

Additional Key Words and Phrases: Method Name Prediction, Naming Convention.

ACM Reference Format:
Shangwen Wang, Ming Wen, Bo Lin, Yepang Liu, Tegawendé F. Bissyandé, and Xiaoguang Mao. 2023. Pre-Implementation Method Name Prediction for Object-Oriented Programming. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (April 2023), 35 pages. https://doi.org/10.1145/xxxxxxx.xxxxxxx

1 INTRODUCTION

During software development, developers often need to think about which methods to implement after defining a class. Devising high-quality names for these methods, which are often representatives of the class's functionalities, is important. Unfortunately, method naming can be challenging for developers [52, 53, 64], in particular when these developers are inexperienced or unfamiliar with the project [21, 48, 55]. It is therefore desirable to have a tool that can assist developers in recommending names for methods that are going to be implemented in a class.

A number of approaches have been proposed in recent literature to automate the task of method name recommendations [12, 17, 26, 58, 69, 86]. With MNire [69] and Cognac [86], researchers proposed to translate the token sequences extracted from program entities into method names. In another direction, code2vec [18] and code2seq [17], which leverage the structural information in the Abstract Syntax Tree (AST) representation of a program, have been used to recommend high-quality method names. Despite the high performance achieved by these approaches, their application is often limited to predicting mistakes in the names of implemented methods. Indeed, literature approaches often explicitly assume that the method body is already available and therefore leverage it as an input in the method name recommendation process. Such an assumption is impractical in recommending method names during the software development phase where signature definition (including method naming) precedes method implementation [1, 3]. Consequently, current method name recommendation approaches are more suitable to support software evolution (e.g., for method name refactoring) rather than facilitating initial software development. On the other hand, the usefulness of popular Integrated Development Environments (IDEs) such as Eclipse [2] is also compromised as they can only recommend simple method names (e.g., setters/getters).

The aforementioned shortcomings in both research and practice motivate our investigation into the problem of *predicting names for all methods that are likely to be implemented within a newly defined class*. We refer to this new software automation task as "**Pre-Implementation Method Name Prediction (PI-MNP)**". To the best of our knowledge, we are the first to address this task in the literature.

To better understand developers' practices when implementing methods, we performed an exploratory survey with 101 software practitioners from 19 countries across five continents. Our survey shows that (1) more than 90% of the respondents define method signature first when implementing methods, thus supporting our core hypothesis for this work; and (2) more than 70% of the respondents perceive the usefulness of recommending method names before implementation, highlighting the relevance of the newly-defined automation task. Concretely, realising PI-MNP has two potential benefits. First, it can save developers' efforts in designing and implementing a class. Second, it can help improve software quality since existing literature findings suggest that providing developers with hints on the functionalities that should be implemented can help avoid bugs [39].

A prerequisite to perform the PI-MNP task, however, is to understand the class's purposes (i.e., the developers' potential intentions for this class), which is challenging considering that we only have limited information at hand when the class is first defined. We rely on the class names to identify such purposes since they often reflect the core concepts implemented in the source code [29, 33, 73]. Under this assumption, we can make several observations (cf. Motivating Example in Section 3). First, the names of some methods relate to specific class fields, whose naming practices should also take the field information into consideration. Therefore, we are motivated to classify the method names into two types (i.e., those related to class fields and those independent from class fields) to deal with them separately. Second, method names within a class are highly correlated with those from *proximate classes* (i.e., those classes that are semantically similar to the target class with respect to their names), no matter whether the method names are related to class fields or not. Motivated by these observations, we further performed a large-scale empirical analysis on 430 well-maintained GitHub repositories in pursuit of designing a more effective approach. The empirical results show that for a specific class (1) the tokens of method names that are independent from the fields can be frequently observed from those of its proximate classes; and (2) the method names that are related to class fields largely overlap with those of its proximate classes if they target the same fields. Such results inspire us to adopt different strategies to predict these two types of names. First, for methods that are independent from the class fields, we need an inference model to predict the tokens composing the method names. Indeed, prior work has shown that method names do not tend to repeatedly occur among projects [69], but their constituting tokens can be readily inferred [58, 69, 86]. Second, for names related to the class fields, we can design a deterministic method since our second observation suggests that such names from proximate classes are highly similar to the ones that are required to be implemented in the target class.

Supported by our empirical findings, we propose a Method nAme pRedIctOr, `Mario`, to predict the names of all the methods that are going to be implemented in a class mainly based on the class name. Specifically, `Mario` utilizes a large source code corpora (a.k.a "Big Code" [14, 45, 54]) where abundant proximate classes can be found for target classes. `Mario` can thus learn diverse and rich knowledge from the codebase to make predictions for different types of method names. Specifically, for methods independent from class fields, `Mario` utilizes a deep learning model, Transformer [84], to infer their names with the help of those methods from the proximate classes. This design choice is inspired by prior studies [58, 69, 86], which have shown that the method name can be effectively translated by a sequence of tokens where a large proportion of its composing tokens are involved. Upon provided with a number of fields, `Mario` will further investigate if the field can be observed from the proximate classes of the target one. If so, it directly reuses the method names from the proximate classes that are related to the field. Such a decision is supported by our empirical findings. Otherwise, a set of pre-collected *prior knowledge* is utilized to decide what names should be predicted for it.

To evaluate the effectiveness of our approach, we performed extensive experiments on 22,822 classes from 300 top-starred Java projects in GitHub. `Mario` can be set with several configurations according to the desired sensitivity in identifying proximate classes. When the sensitivity is high, the requirement for identifying a proximate class in the codebase is less rigid, and thus `Mario` can work for around 90% of the classes with an F-score reaching 45% at the token level (i.e., comparing the tokens composing the method names with the ground truth). In contrast, when the sensitivity is low, `Mario` is usable for around 40% of the classes, but the F-score at the token level can go beyond 60%, which slightly exceeds that of the state-of-the-art code2seq [17] (whose predictions further relies on method implementation details). Additionally, the performance scores of `Mario` are significantly higher than those of three rule-based baselines that we have designed. For instance, with the intuition that there may be class pairs with high similarities due to code clone, a designed

baseline is to reuse method names from the class whose name is the most similar with the target class. Results show that `Mario` can outperform this baseline by at least 20% with respect to the F-score at the token level.

In summary, our study makes the following major contributions:

- **Significance:** We target a challenging task, PI-MNP, which is in line with developers' needs based on their coding practices. Realising this task has a great potential to boost software development productivity.
- **Empirical results:** Our study deepens our community's understanding of the relationship between the method names of a class and those of semantically-related classes with similar names.
- **PI-MNP with `Mario`:** Supported by our empirical findings, we implement `Mario`, a recommender system to predict the names of methods that are going to be implemented within a class. The approach mainly relies on the class name. The experimental results have demonstrated the effectiveness of `Mario`.
- **Open science policy:** We open source the materials in this study at **https://github.com/ShangwenWang/Mario** to facilitate replications and follow-up studies.

2 DEVELOPERS' PERSPECTIVES ON THE PI-MNP TASK

To further motivate this study, we first conducted an online survey to understand the developers' coding practice and perspectives on the PI-MNP task.

Respondents. To obtain a sufficient number of respondents from diverse background, we followed a multi-pronged strategy to recruit respondents. We first sent emails to our contacts at four top-tier IT companies in China (which are ByteDance, Baidu, Alibaba, and Tencent, respectively) and asked them to disseminate our survey. Finally, we received 61 responses from these four companies. We then sent emails with a link to the survey to 609 Java&Android developers from GitHub projects, aiming to recruit open-source developers working in the software industry. We obtained 40 responses out of these emails (a response rate of 6.6%). In the end, we obtained 101 responses. These respondents reside in 19 countries across five continents. The top two countries in which the respondents reside are China (61) and the United States (14). The respondents have an average of 7.2 years of professional experience in software development (min: 0.5, max: 36, median: 5, sd: 6.8).

Question#1. The first question focuses on the developers' coding practice. Specifically, we ask them "When writing a method, do you define the signature first or implement the body first?". The respondents are asked to select the answer from "Define the signature first and then implement the body" and "Implement the body first and then fulfill the signature". From the answers, 92% of the respondents (i.e., 93 in total) reveal that they define the signature first and then implement the body while only 8 respondents choose to implement methods in the reverse order.

Question#2. The second question focuses on the developers' perspectives on the PI-MNP task. Specifically, we ask them "Would it be useful for your software development if there is a tool (or IDE plugin) that can recommend the names for all methods that are likely to be implemented within a newly defined class?". The respondents are asked to assess the usefulness on a 5 point Likert scale and one more extra option (Very Unuseful, Unuseful, Neutral, Useful, Very Useful, and I Don't Know). The "I Don't Know" option was provided in case the respondents had a poor understanding of the statement.

Figure 1 shows the respondents' ratings of the usefulness of recommending all the names for methods that will be potentially implemented in a class. We totally obtained 98 valid responses (i.e., three respondents chose "I Don't Know"). From the results, 46% of the respondents (45) consider the

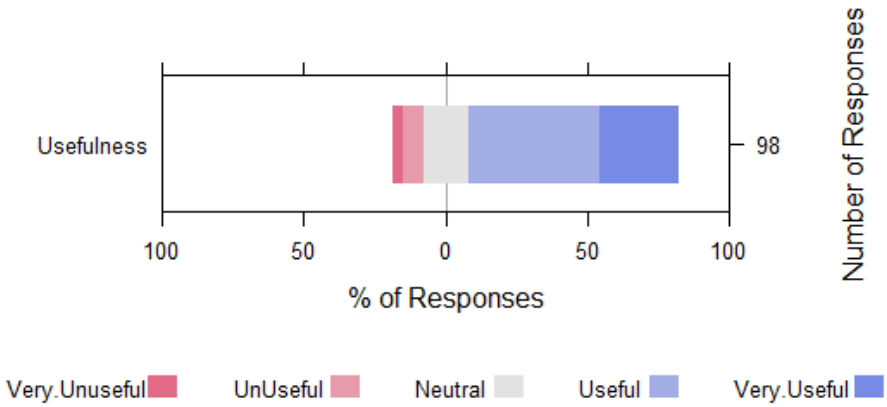


Fig. 1. Results of the second question in our survey.

```
1|  |class ZipOutputStream {
2|  |
3| ✗| private byte[] comment;
4| ✔| private int method;
5| ✔| private boolean finished;
6| ✗|
7| ✗|
8|  |
9| ✔| public void putNextEntry(){...}
10| ✗| public void writeBytes(){...}
11| ✗| public void writeExtra(){...}
12| ✗| public void getExtraLen(){...}
13|  |
14| ✔| public void setComment(){...}
15| ✔| public void setMethod(){...}
16| ✗|
17| ✗|
18|  |}
```

(a) the ZipOutputStream.java from the *j2objc* project

```
1|  |class ZipOutputStream {
2|  |
3| ✗| private String comment;
4| ✔| private int method;
5| ✔| private boolean finished;
6| ✗| private String encoding;
7| ✗| private long cdOffset;
8|  |
9| ✔| public void putNextEntry(){...}
10| ✗| public void getBytes(){...}
11| ✗| public void getZipExtra(){...}
12| ✗|
13|  |
14| ✔| public void setComment(){...}
15| ✔| public void setMethod(){...}
16| ✗| public void setEncoding(){...}
17| ✗| public void getEncoding(){...}
18|  |}
```

(b) the ZipOutputStream.java from the *Apache Ant* project

Fig. 2. Two classes with the same name from different projects. A check (cross) mark denotes the corresponding contents from the two classes are identical (different).

recommendation task *useful* and 28% of the respondents (27) consider it *very useful*. In contrast, only 3 and 7 respondents consider the recommendation *very unuseful* and *unuseful* for their development, respectively.

Our survey results reveal that (1) the majority of developers (more than 90%) define the signature first when writing methods; and (2) most of the developers (around 70%) consider the PI-MNP task is useful or very useful for their development. Therefore, performing the PI-MNP task has the potential to be useful in practice.

3 MOTIVATING EXAMPLE

To perform the PI-MNP task, one needs to understand the intentions of the class. Utilizing the class documentation is impractical since, usually, the class documentation is automatically generated after the implementation of the class source code [44, 67]. On the contrary, we rely on analyzing the class names since class names represent the core concepts encoded in Object-Oriented source code [29, 33]. Our motivation is shown in Figure 2 where the enclosing contents of two classes with the identical class name are briefly illustrated. Usually, the contents of a class can be split into two parts: the fields, and the methods [5]. Besides, we observe that the methods can be further separated into two types: those related to its fields, and those independent from its fields. The differences between the two types of methods can be reflected by their names, that is, whether their names are correlated with a specific class field. For instance, the method `putNextEntry` relates to no field while the method `setMethod` obviously relates to the field `method` in the example as shown in Figure 2.

Through this example, we mainly gain the following observations. ❶ The two classes tend to possess similar names for the methods that are independent from class fields. Listed from line 9 to line 12, some of the method names are identical (e.g., both two classes possess a method named `putNextEntry`) while the others are composed by similar tokens even if they are not identical (e.g., the tokens `get` and `extra` occur in both `getExtraLen` and `getZipExtra`). This indicates that for classes with similar names, the names of their methods that are independent from the fields may share similar tokens. ❷ For the shared fields (e.g., `method`) which are possessed by both classes, the classes might contain methods with identical names that are related to them (i.e., `setMethod`). Listed from line 14 to line 17, we note that the two methods only possessed by the class at the right side (i.e., `setEncoding` and `getEncoding`) are related to its unique field (i.e., the `encoding` which is not in the class at the left side). This indicates that if two classes possess identical fields, they tend to possess methods with identical names that are related to these fields. Be noted that we can only focus on the name of the fields when deciding if they are identical among classes. The behind intuition is that names can already represent the semantic information of identifiers well [7, 8, 24]. Taking line 3 in Figure 2 as an example, although the two fields have different data types (`byte[]` vs. `String`), they can be considered as identical and the classes possess two methods with identical names for them respectively (cf. line 14). Another notable phenomenon is that for fields that are uniquely possessed by a class, some of them may still have related methods. For instance, the class at the right side has a field `encoding` that does not occur in the class at the left side while this field has two related method names (e.g., `getEncoding`). This indicates that to perform the PI-MNP task, we also need to recommend names for certain fields that are uniquely possessed by a class.

From the above analysis, we find that for classes whose names are similar, the names of their methods, no matter whether they are related to the fields, may be similar to some extent. Therefore, we are motivated to utilize classes with similar names to infer the method names for a specific class.

4 EMPIRICAL INVESTIGATION

Inspired by the motivation example, we further performed an empirical study to investigate whether such observations are pervasive among real-world projects.

Dataset. To perform the empirical analysis, we used the dataset collected by Liu *et al.* [64], which contains 430 well-maintained and open-sourced Java projects. In our study, we focused on the source code to reduce potential bias from the test code. Therefore, we totally analyzed 258,321 classes. Note that we omitted constructors and *main* methods within the classes during our empirical analysis as well as the evaluation, since such methods do not require any prediction.

4.1 Definitions

To ease our presentation in the rest of this paper, we define several concepts here:

- **Semantic class name:** denotes the last four words of the full qualified class name (e.g., org.apache.commons.fileupload.util.mime.ParseException.java). We consider the full qualified name since the name of the package where the class is located can help understand the functionality and intention of the class. For instance, the word `util` can indicate the class implements some utility functions that can be invoked by other classes. We only consider the last four words (split by ".") since our statistic on the empirical dataset shows that the median number of words in full qualified names is 7 which means the last four words are sufficient to cover the semantics considering the first three usually denote organizations (e.g., org.apache.commons), providing useless information about the semantic context of the class.
- **Proximate class pair:** denotes a pair of classes that satisfy two conditions: ① the rightmost tokens of their class names are identical; and ② the semantic similarity between their semantic class names exceeds a threshold (i.e., α, which is calculated by a customized fastText model [41]. Details will be introduced later in Section 4.2). We set the above two conditions due to the following reasons. First, Butler *et al.* [29] observed that class names often end up with a noun with respect to the grammar structure, while Singer and Kirkham [79] showed that this noun is usually an indicator of the programmer's expectation of the class. These studies suggest that the rightmost noun in a class name can potentially indicate the functionality the class is expected to implement. We, therefore, set the first condition. We set the second condition to further ensure that the involved two classes are semantic-related. A strict restriction that can ensure a proximate class pair is strongly related would be to require the class names to be identical. However, we performed a preliminary study on the class names and found that only around 10% of the unique class names occur more than once in our dataset. It means that if the strict condition is applied, very few of the classes can find their proximate classes. Consequently, there is no sufficient dataset to guarantee the effectiveness and generality of our approach. We, therefore, set the above conditions which are more flexible by calculating the semantic similarity. In the rest of this paper, a class's proximate classes denote those classes that satisfy the above two conditions with the target one. It should be noted that our definition of "proximate class" in the paper only focuses on the semantic similarity between class names, and it is not necessarily related to the class body in terms of functionalities.
- **Field-relevant method:** denotes a method whose name is related to a field within the class. Formally, given a class field *f*, the name of the method related to *f* can be split as: $V + f$ where V denotes a verb. A simple way to judge if a method's name is related to a field is to check if the field is a sub-string of the method's name. However, we find that developers sometimes define a general field as well as a more specialized field in the same class (e.g., *path* and *detailPath*). Under such conditions, simply applying the above process will introduce noises (e.g., relating the method whose name is *getDetailPath* with the field *path*). Our definition is based on our observation that the name of a method relating to a class field often starts with a verb (e.g., `set` and `get`), indicating the operation in this method, which is then followed by the associated field name. Our definition is also supported by several previous studies. For instance, Abebe *et al.* [6] and Gupta *et al.* [42] assert that method names should be verb phrases, while Butler *et al.* [28] show that a large proportion of field names are noun phrases. Therefore, considering that a method name is composed of a verb and a field name is, at least, consistent with the naming conventions.
- **Field-irrelevant method:** denotes a method whose name cannot be matched with any field in the class based on the above-mentioned rule.

- **Unique fields:** denote those fields in a class that do not occur in any of its proximate classes (e.g., *encoding* for the class at the right side in Figure 2).

4.2 Experiment Setting

Recall that our empirical analysis aims to investigate whether the two observations in Section 3 are pervasive. Therefore, we design the following research questions:

RQ1: How pervasive are proximate class pairs?

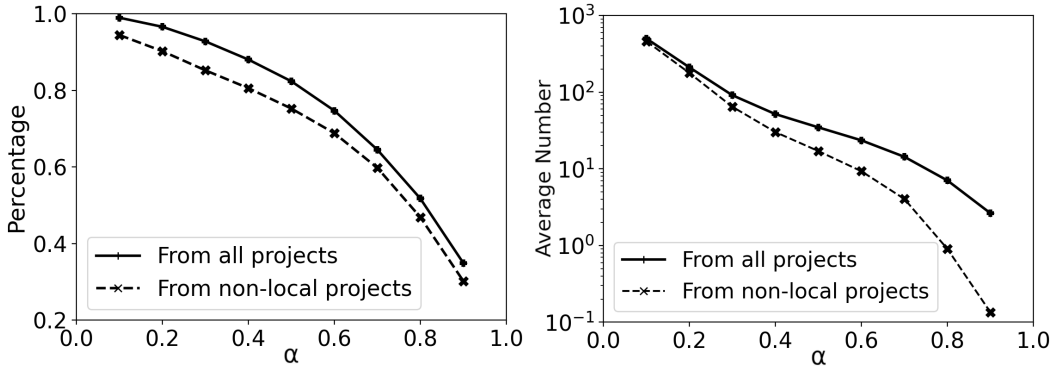
RQ2: How is the relation between the field-irrelevant method names of a class and those of its proximate classes?

RQ3: How is the similarity between field-relevant method names of a class and those of its proximate classes with respect to its non-unique fields?

Suppose that we are going to perform the PI-MNP task with the help of proximate classes, one essential question is how likely there exists proximate classes for reference, which will be answered by RQ1 via investigating the pervasiveness of proximate class pairs. Furthermore, how to utilize the information from proximate classes for inferring the field-(ir)relevant method names of the target class also needs investigation: for an effective approach, we need to understand do we need to reorganize the information from proximate classes or can we directly reuse them. The designed RQ2 and RQ3 exactly aim to answer such critical questions for the approach design. Specifically, RQ2 investigates to what extent the field-irrelevant method names of a class appear in those of its proximate classes, whose results can guide the approach design for predicting filed-irrelevant method names. For instance, if results reveal that a large number of field-irrelevant method names of a class appear in those of its proximate classes, we may design a reuse-based heuristic for predicting field-irrelevant method names; otherwise we may need to reorganize the field-irrelevant method names from proximate classes. Similarly, RQ3 investigates to what extent the field-relevant method names of a class appear in those of its proximate classes, whose results can guide the approach design for predicting filed-relevant method names. Overall, the answers to our research questions provide empirical foundations on the pervasiveness of proximate class pairs, and the predictive capabilities of proximate classes on field-(ir)relevant method names of the target class. Such foundations can shed lights on how to design effective approaches for predicting method names.

Data processing. To represent the semantic similarity of two class names, we embed the names as vectors and then calculate the cosine similarity as an indicator of their semantic similarity. Existing pre-trained language models (like BERT [34]) are learned from natural language corpus, where the co-occurrence relationships among tokens significantly differ against those from full qualified class names. We thus chose to train a fastText model [41] from scratch on all the full qualified class names within our dataset. Note that during training, the class names, which are usually composed of several tokens, are split based on the *camel case* and *underscore* naming convention while other words within the full qualified class names (whose letters are usually all in lower case, e.g., *fileupload* in the example in Section 4.1) are split with the help of Word Ninja.[1] After training, the last four words from the full qualified class name are considered, and we separately embed the class name (i.e., the last word) and its context information (i.e., the first three words). Specifically, the class name composed of n tokens is embedded as $c_1, c_2, \ldots, c_n$, and the average $\overline{c}$ is used to represent the class name. While the other three words are split into tokens $t_1, t_2, \ldots, t_m$ whose embeddings are $d_1, d_2, \ldots, d_m$. Their average value $\overline{d}$ is used to represent the context information. At last, the two contexts are concatenated to represent the semantic of the class name as $F = [\overline{d}; \overline{c}]$. In our experiment, the dimension of each embedded token is 128.

[1] https://github.com/keredson/wordninja



(a) The correlation between P_c and the threshold α. **(b)** The correlation between Ave and the threshold α.

Fig. 3. The values of two indicators under different threshold (α).

In our study, to determine if a method name token is a verb or not, we obtained the part-of-speech (PoS) information of each token in the method name by following previous studies [71, 90]. Such a process requires to (1) convert each method name to a sentence by splitting the name into tokens and prepending the result with the token "I", and (2) apply the Stanford Tagger [4] to obtain the results.

It should also be noted that when analyzing class fields, we discarded constants. This is because that constants are immutable and can be easily accessed. Thus, it is unlikely that a method name would be related to the constants. Any field whose characters are all in the upper case is identified as a constant, following the official naming conventions of Oracle.[2]

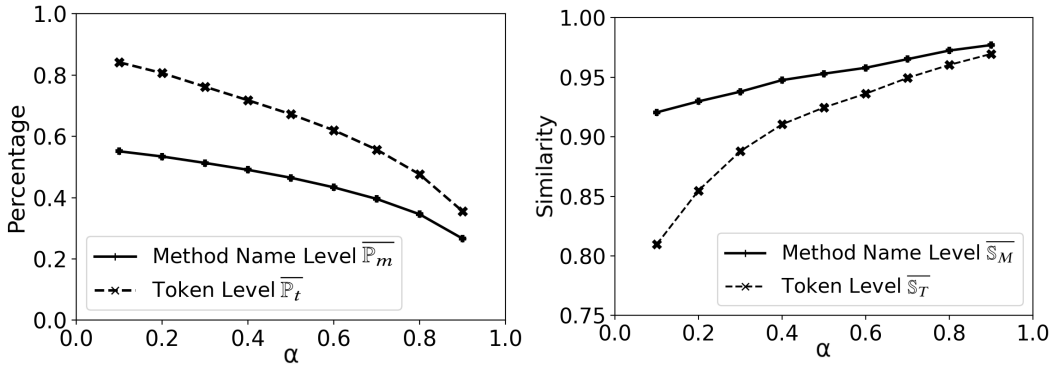
4.3 Results

4.3.1 RQ1: the abundance of proximate class pairs. In this RQ, we used two indicators to approximate the abundance of proximate class pairs under diverse values of α. The first one is the percentage of the classes that have at least one proximate class (P_c) while the second one is the average number of proximate classes a certain class can possess (Ave). These two indicators reflect the abundance of proximate class pairs from different perspectives. For a specific class, we also dissected the origin of its proximate classes. Specifically, we investigated if the proximate classes of the target one are from the same project. Therefore, the results are analyzed from two granularities: considering proximate classes from all investigated projects (including those from the same project as the target one) and considering proximate classes from non-local projects (explicitly excluding those from the same project as the target one). While the former can be applied when a part of the current project is implemented, the latter provides a more general perspective. Results are shown in Figure 3.

From the results, we find there are abundant proximate classes for a specific class. For instance, if we consider all the projects, more than 80% of the total classes can find their proximate classes and one class can possess around 35 proximate classes on average when α is 0.5. We also note that explicitly considering non-local projects can already provide abundant proximate classes. For instance, 76% of the total classes can find their proximate classes when α is 0.5, slightly lower than the percentage obtained when local projects are considered (i.e., 82%).

4.3.2 RQ2: the relationship between the names of the field-irrelevant methods of a class and those of its proximate classes. Inspired by our motivation example, we postulate that the names of the field-irrelevant methods of a class can be related to those of its proximate classes. We therefore investigated the extend to which the contents of a class's field-irrelevant method names appear in

[2] https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html



(a) The correlation between $\overline{\mathbb{P}_m}$ & $\overline{\mathbb{P}_t}$ and the threshold α.

(b) The correlation between $\overline{\mathbb{S}_M}$ & $\overline{\mathbb{S}_T}$ and the threshold α.

Fig. 4. The values of four indicators under different threshold (α).

those of its proximate classes. The analysis is based on two granularities: the method name level and the level from the tokens composing the method names (so called token level). Suppose the field-irrelevant methods within the class are $\mathbb{M}_1 = \{m_1, m_2, \ldots, m_i\}$, the field-irrelevant methods from all of its proximate classes are $\mathbb{M}_2 = \{m_1^{'}, m_2^{'} \ldots, m_j^{'}\}$. The method name level result is calculated as $\mathbb{P}_m = \frac{|\mathbb{M}_1 \cap \mathbb{M}_2|}{|\mathbb{M}_1|}$ which is the percentage of $\mathbb{M}_1$'s methods that occur in $\mathbb{M}_2$. By splitting method names into tokens, we can gain $\mathbb{T}_1$ which denotes the set of tokens from the methods within $\mathbb{M}_1$ and the same for $\mathbb{T}_2$. Then the token level result is calculated as $\mathbb{P}_t = \frac{|\mathbb{T}_1 \cap \mathbb{T}_2|}{|\mathbb{T}_1|}$. After obtaining the values of $\mathbb{P}_m$ and $\mathbb{P}_t$ for each class, the overall values ($\overline{\mathbb{P}_m}$ and $\overline{\mathbb{P}_t}$) on the whole dataset are calculated as the averages.

The results are shown in Fig. 4a. We mainly note two phenomenons. First, in general, the method name level overlap between a class's field-irrelevant methods and those of its proximate classes is high, which confirms our observation. Specifically, the $\overline{\mathbb{P}_m}$ is around 0.45 when α is 0.5. Second, the token level overlap is even higher than that of the method name level. For instance, when α is 0.5, the $\overline{\mathbb{P}_t}$ nearly reaches 0.7. Such results are consistent with the observations from Nguyen *et al.* [69] that the method names tend to occur more uniquely than the corresponding tokens. This suggests that the token level information of the field-irrelevant method names from a class's proximate classes may provide more powerful effects for predicting the field-irrelevant method names for the target class compared with directly using the method name level information. Nonetheless, we notice that when α is 0.5, a class possesses 35 proximate classes on average, which could lead to a large-size of $\mathbb{T}_2$. Therefore, when utilizing such information, a challenge that needs to be addressed is how to search for the target tokens effectively. We propose to utilize deep learning techniques to address such a challenge (see Section 5).

4.3.3 RQ3: the similarity between the names of the field-relevant methods of a class and those of its proximate classes with respect to their overlapped fields. Motivated by our observation, we postulate that for the non-unique fields of a class, the field-relevant method names from this class have high chance to be identical with those from its proximate classes. We, therefore, proposed to analyze the similarity between the field-relevant method names of a class with respect to its non-unique fields and those from its proximate classes. Specifically, given a class C and its proximate classes $\mathbb{C}_p = \{C_1, C_2, \ldots, C_m\}$, the non-unique fields of C are denoted as $O_f = \{f | (f \ in \ C) \cap (\exists C^{'} \in \mathbb{C}_p, f \ in \ C^{'})\}$, where $f \ in \ C$ denotes f is a field of the class C. Then, the field-relevant methods of C related with fields from O_f are $\mathbb{FR}_M$ while those of $\mathbb{C}_p$ are $\mathbb{FR}_{M_p}$. Tokens composing the methods within $\mathbb{FR}_M$

are denoted as $\mathbb{FR}_T$ and those composing the methods within $\mathbb{FR}_{M_p}$ are denoted as $\mathbb{FR}_{T_p}$. The method name level similarity is calculated as $\mathbb{S}_M = \frac{|\mathbb{FR}_M \cap \mathbb{FR}_{M_p}|}{|\mathbb{FR}_M \cup \mathbb{FR}_{M_p}|}$, which is the Jaccard similarity of the two sets, while the token level similarity (i.e., $\mathbb{S}_T$) is the Jaccard similarity between the sets $\mathbb{FR}_T$ and $\mathbb{FR}_{T_p}$. Finally, the values on the whole dataset ($\overline{\mathbb{S}_M}$ and $\overline{\mathbb{S}_T}$) are the averages of those of all classes in the dataset. As discussed in Section 3, we only consider the field name to judge if a field is unique.

Results are shown in Fig. 4b. We observe that the values of $\overline{\mathbb{S}_M}$ and $\overline{\mathbb{S}_T}$ are both extremely high. Specifically, the similarity of the method name level ($\overline{\mathbb{S}_M}$) exceeds 0.9 at all time while the similarity of token level ($\overline{\mathbb{S}_T}$) also exceeds 0.9 when α is higher than 0.5. Another interesting finding is that $\overline{\mathbb{S}_M}$ is higher than $\overline{\mathbb{S}_T}$. Generally speaking, this happens because tokens composing the investigated method names are repetitive. Therefore the sizes of $\mathbb{FR}_T$ and $\mathbb{FR}_{T_p}$ are smaller than those of $\mathbb{FR}_M$ and $\mathbb{FR}_{M_p}$. Then when an uncommon token is introduced, which only leads to one uncommon method, the decrease with respect to $\overline{\mathbb{S}_T}$ is more significant compared with that of $\overline{\mathbb{S}_M}$. We give a concrete example here. For the `ErrorsTag.java` class of the *Apache struts* project, [3] it contains 12 field-relevant method names obtained by combining the Verbs = {get, set} and the Fields = {bundle, footer, locale, name, property, header} in pairs. When $\alpha = 0.5$, its proximate classes totally have 14 field-relevant method names composed by the above 12 ones plus with `prepareName` and `createLocale`. Under such a condition, the Jaccard similarity of the method name level is 0.857 (12/14), higher than that of the token level which is 0.8 (8/10).

Overall, our results indicate that if a class and its proximate classes contain identical fields, they tend to contain the same methods that are related to these fields.

We conclude the main findings obtained through our empirical analysis as following:

[F1] *Proximate class pairs are pervasive among real-world projects.*
[F2] *A considerable percentage of tokens composing the names of a class's field-irrelevant methods can be found in the names of its proximate classes' field-irrelevant methods.*
[F3] *The similarity between a class's method names of its non-unique fields and those of its proximate classes is extremely high.*

In this study, we use distributed representations to model semantics and similarities of class names. Therefore, it is hard to ensure that the retrieved proximate class pairs possess identical intentions. However, according to our empirical results that method names from proximate class pairs overlap to a certain degree, we presume that the retrieved proximate class pairs, at least, have similar semantics.

5 METHODOLOGY

5.1 Overview

Figure 5 illustrates the overall workflow of `Mario`, which is a hybrid approach for method name prediction.

Given a class with a number of fields named, `Mario` first searches for the proximate classes within a large *class repository* and the classes from the local project (i.e., the project which contains the target class). The selection process is described in Section 4.2: a class whose similarity with the target class regarding to their semantic class names exceeds the threshold α will be included. After that, `Mario` adopts a number of different strategies to generate method names for the target class. For field-irrelevant method names, directly reusing those from the proximate classes is

[3] https://github.com/apache/struts

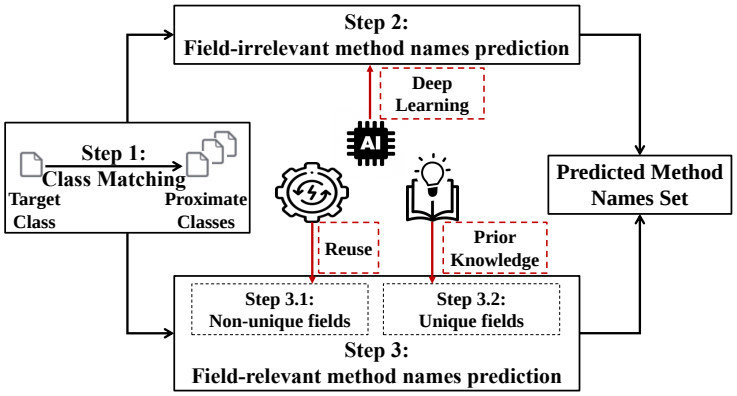


Fig. 5. The workflow of `Mario`.

inappropriate since the previous study [69] shows that only one third of the method names occur for multiple times among a large-scale software repository and our further investigation reveals that this conclusion also holds for field-irrelevant methods. Fortunately, our investigation has revealed that the field-irrelevant method names of a class and those of its proximate classes, although not highly identical, have great correlations with respect to the tokens composing them (**Finding-2**). Therefore, for predicting field-irrelevant method names, the challenge is how to search for the target tokens from a large number of candidate tokens as we have mentioned in Section 4.3. Manually summarizing patterns is unsuitable considering the diversity of method name tokens. On the contrary, supported by several previous studies [58, 69, 86] where the tokens composing the method names are automatically inferred from a large amount of inputs by deep learning techniques, we are motivated to address this challenge by utilizing the power of deep learning. In particular, we use a model, named Transformer [84], to transform the tokens composing the field-irrelevant method names of the proximate classes into the tokens composing the field-irrelevant method names of the target class. For field-relevant methods, the basic idea is the approach should work for each individual field, since the set of class fields is usually expanded gradually with the development process. To that end, we consider if the fields are unique (i.e., cannot be observed from the proximate classes) or not (i.e., can be observed from the proximate classes). For fields that are non-unique, we directly reuse all the method names from the proximate classes that are related to these fields. This decision is based on our investigation that for fields that can be commonly observed in a class and its proximate classes, they tend to share very similar field-relevant method names (**Finding-3**). For fields that are unique, we need to decide which verbs should be combined with them to shape the corresponding field-relevant method names. To achieve so, we performed empirical investigation on the dataset and summarize *prior knowledge* to help us make such decisions. After the above procedures, the prediction results from the individual parts are integrated as the final results of `Mario`.

5.2 Transformer Based Field-Irrelevant Method Names Prediction

In our study, we use a sequence-to-sequence (seq2seq) model to infer tokens of the field-irrelevant method names by the tokens composing the field-irrelevant method names of the proximate classes. We decide to adopt a Transformer model [84] since it achieves the state-of-the-art performances on a number of seq2seq tasks [9, 47, 56]. Several studies have investigated the semantic patterns of Java method names and found that the majority of them are verb phases [6, 24, 42]. Therefore, from the perspective of natural language processing (NLP), the rationale of this decision is that we can train a translator which translates a series of verb phrases into another series of verb phrases.

Primer on Transformer. We briefly introduce Transformers here. Transformers, with the widely-known Encoder-Decoder architecture, are designed for dealing with sequence data. The input sequence $[i_1, i_2, \ldots, i_L]$ will be embedded as $\mathbf{X} = [x_1, x_2, \ldots, x_L]$, $x_i \in \mathbb{R}^{d_{model}}$ firstly. The key point of the model is the mechanism called *self-attention*, which maps the embedding sequence to a sequence of the same length $[z_1, z_2, \ldots, z_L]$, $z_i \in \mathbb{R}^{d_z}$. To achieve so, $\mathbf{X}$ is fed to three fully-connected networks ($\mathbf{W_q}$, $\mathbf{W_k}$, $\mathbf{W_v}$) to calculate the query, key, and value vectors:

$$\mathbf{Q} = \mathbf{X}\mathbf{W_q}, \quad \mathbf{K} = \mathbf{X}\mathbf{W_k}, \quad \mathbf{V} = \mathbf{X}\mathbf{W_v} \tag{1}$$

The output is calculated as the weighted combination:

$$Attn(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V} \tag{2}$$

where d_k is the dimension of the key vector.

Several (h) attention layers (so called heads) are applied in parallel and a Transformer block is constituted by the multi-head attention, a residual connection, a layer normalization, and a position-wise fully connected feed-forward network. In our study, for the Encoder and Decoder, the numbers of blocks are both six while for each block, the number of attention layers (h) is set to eight. The values for hyper-parameters of the model are reused from the open-source implementation of Transformers.[4] For more details about the Transformer model, readers can refer to [56, 84].

Inputs of our model. We recall that `Mario` utilizes the information of field-irrelevant method names from all the proximate classes. From our empirical analysis, a class may possess hundreds of or even thousands of proximate classes, which will lead to the input of our model being too long. To avoid such a case, we need to prioritize these field-irrelevant method names. Specifically, we design the following heuristics to rank the method names. Method names are first ranked in a descending order based on their numbers of occurrences in all the proximate classes. For names that occur for the same times, those from the proximate class which possesses higher semantic similarity with the target class are ranked in priority.

After obtaining the rankings for all the method names, `Mario` selects the top 10 ones. This number is empirically determined. Specifically, we also experimented with other choices such as 5, 20, 50 and etc. , but found the effectiveness decreased under all such conditions. Indeed, if it is too small, the input may provide limited information; while if it is too large, the task we target is more likely to be text summarization, where distinguishing the more informative parts of the input is a long-standing challenge [43, 78]. The selected method names are then split into tokens based on the camel case and underscore naming conventions, and all obtained tokens are transformed in to their lowercase form. We use a special token "," to separate individual method names. Let us use Figure 2 as an example. Suppose the class at the right hand is the only proximate class in the code corpora of the class at the left hand, the input token sequence of the model is "put next entry , get bytes , get zip extra", and the oracle output is "put next entry , write bytes , write extra , get extra len". After training, we can obtain the predicted method name list via splitting the output sequence by ",".

5.3 Prior Knowledge Based Field-Relevant Method Names Prediction for Unique Fields

In our study, we summarize properties for class fields occurred in our empirical dataset in order to predict method names for a class's unique fields. Such field properties serve as the *prior knowledge* of our approach, which will be introduced in detail as follows.

[4] https://github.com/SamLynnEvans/Transformer

To recommend method names for a specific field, we need to leverage the knowledge from the existing fields that are similar to it. Similar to the method names and class names, it is rare to observe two fields with identical names. We therefore consider two fields are similar if their rightmost tokens are identical. That is to say, we categorize the fields in our study based on their rightmost tokens. For instance, both the fields *detailedPath* and *abstractPath* will be classified into the cluster *path*. The behind intuition is that the previous study [28] shows that the overwhelming majority of field names in Java are noun phrases (e.g., *formatData*). We thus postulate that the rightmost token can serve as the representation of the semantic meaning of the field, following the existing study on class names [79]. By doing so, after encountering a previously unseen field, we can utilize the knowledge from its cluster to make the prediction.

For each unique field, we need to decide which verbs will be used to combine with it to construct the field-relevant method names. To achieve so, we summarize the probabilities of each field to occur in the method names with different verbs and then recommend the verbs with high probabilities. Formally, for the fields whose rightmost token is t, their cluster is denoted as $\mathbb{C}_t = \{f_1, f_2, \ldots, f_n\}$. Suppose $\mathbb{V} = \{verb_1, \ldots, verb_N\}$ represents the verb vocabulary. For a specific field f_i, we record from all the classes within our empirical dataset the number of classes that possess f_i (denoted as $NCP(f_i)$). For each verb $verb_j$, we also record the number of classes that possess field-relevant method names composed by f_i and $verb_j$ (denoted as $NCF(f_i, verb_j)$). Then, the probability of $verb_j$ to combine with fields from $\mathbb{C}_t$ is calculated as: $\frac{\sum_{i=1}^{n} NCF(f_i, verb_j)}{\sum_{i=1}^{n} NCP(f_i)}$. Given a unique field, `Mario` can then decide which cluster it belongs to and recommend verbs with high probabilities based on the summarized prior knowledge. Specifically, if the probability of a verb to combine with it exceeds a threshold (β), `Mario` will recommend it to construct a field-relevant method name.

6 EVALUATION

6.1 Research Questions

We seek to answer the following research questions in this study.

RQ4: How is the effectiveness of `Mario`?

RQ5: How is the effectiveness of `Mario` on field-relevant method names and field-irrelevant method names respectively?

In RQ4, we compare the effectiveness of `Mario` with three baselines as well as two state-of-the-art method name recommendation approaches. Moreover, our approach involves a hyper-parameter (α). Through RQ4, we also dissect the effectiveness of `Mario` under diverse values of α. As we have introduced, the method names of a class can be split into two types: field-relevant and field-irrelevant. Apart from the overall effectiveness, we, in RQ5, separately investigate the effectiveness of `Mario` on the two types of method names. Answering such a question can help us understand the strengths and weaknesses of `Mario`.

6.2 Baselines

Note that `Mario` is the first approach that targets PI-MNP. Nonetheless, to show that the design of `Mario` is rational as well as its effectiveness, we develop three baselines based on heuristics and also include two state-of-the-art method name recommendation approaches for comparison.

Baseline#1. Given a specific class, this baseline recommends a setter and a getter method names for each involved field. This baseline mimics conventional IDEs' working mechanism and thus reflects how well the current IDEs (e.g., Eclipse [2]) can perform on the PI-MNP task. This baseline does not require a threshold α since it does not seek for information from proximate classes.

Baseline#2. For a specific class, this baseline always recommends all the method names from the class that is the most similar one to it according to the similarity between their semantic class names.

The rationale of designing this baseline is that there may exist class pairs with high similarities due to code clone. Note that this baseline approach is also independent of the threshold α. It can work as long as it can find another class whose name possesses the same rightmost token with the target class.

Baseline#3. Given a specific class, this baseline recommends the top nine method names that occur the most frequently in its proximate classes. Such a number is selected based on the fact that our empirical dataset reveals that a class contains around nine methods on average. The behind intuition for this baseline is that a method name occurring more frequently in the proximate classes may have a higher probability to be included in the target class, similar to the statistical language models where tokens co-occurring with the target code sequence more frequently are more prone to be predicted [51, 81]. This baseline is affected by the threshold α to select the set of proximate classes.

Code2vec & Code2seq. These are the two state-of-the-art method name recommendation approaches, which rely on the AST structure of programs [17, 18]. Although these two approaches require the method body information, which is not required as the input of `Mario`, we compare `Mario` against them to see how effectively can `Mario` perform on method name prediction.

6.3 Experiment Settings

Dataset. To perform a large-scale evaluation, we chose to use a dataset of 9,550 top-starred Java projects from GitHub, named *Java-large* [17], which has been widely used in evaluating the quality of predicted method names [26, 47, 86]. This dataset has been split into three different parts: 9,000 projects for training, 250 projects for validation, and 300 projects for testing. We decided to evaluate on the test set while searching for proximate classes within the training and validation sets (i.e., the training and validation sets are used as the *class repository*). The inputs of our Transformer model change with the value of α. Therefore, we trained several Transformer models according to different values of α. Note that to avoid data leakage, the Transformer models were trained on our empirical dataset, and we also omitted four projects in the test set that exist in our empirical dataset. We take $\alpha = 0.5$ as an example. Under such a condition, for each class from our empirical dataset, all other classes in this set whose semantic class names are close to that of the target class (i.e., share the same rightmost token with the target one and the semantic similarity to the target one exceeds 0.5) are considered as proximate classes. Then, to construct a training sample, we extract the token sequence of the FI method names of the proximate classes (which is used as the input during training) and the token sequence of the FI method names of the target class (which is used as the oracle during training), as introduced in Section 5.2. The whole training samples are constructed by iterating this process on all the classes in the empirical dataset. After training, the learned Transformer model is applied on the classes from the test set of the *Java-large* dataset. We identify proximate classes from the training and validation sets of the *Java-large* dataset as well as the local project (the criterion of being proximate classes is identical to that in the training process, e.g., the semantic similarity between the semantic class name of a proximate class and that of the target class should exceed 0.5). With the inputs being the token sequence of the FI method names from proximate classes, the model is supposed to predict the FI method names for the target class. Any code token in the test set that does not appear in our training set (i.e., our empirical dataset) is represented as 'UNK'. We investigated that when α equals to 0.5, only around 1.3% (5,927/462,258) of the input tokens are represented as 'UNK'. Therefore, the impact of the Out-of-Vocabulary (OoV) problem on our model is limited.

During the evaluation, we discarded classes that implement interfaces since when developing these classes, developers can directly refer to the methods as listed in the interfaces. The total

number of classes used in our evaluation is 22,822 and the number of classes in our *class repository* is 1,539,568 (1,506,706 from the training set and 32,862 from the validation set).

Metrics. We assess the effectiveness of Mario by computing the following metrics:

***Recall*$_C$:** $Recall_C = \frac{Cla_p}{Cla_t}$ where Cla_p is the number of classes for which Mario can make any prediction independent of its correctness and cla_t is the number of classes within our test set. $Recall_C$ indicates in how many classes Mario could be potentially useful for developers.

***Precision*$_M$, *Recall*$_M$, *F-score*$_M$:** For a specific class whose contained methods are named as $ora_M = \{ora_1, \ldots, ora_n\}$ while the prediction result from Mario is $pre_M = \{pre_1, \ldots, pre_m\}$. Its precision, recall, and F-score are calculated as: $precision = \frac{|ora_M \cap pre_M|}{m}$, $recall = \frac{|ora_M \cap pre_M|}{n}$, $F\text{-}score = \frac{2 \times precision \times recall}{precision + recall}$. Then the performances on the whole dataset ($precision_M$, $recall_M$, $F\text{-}score_M$) are computed as the average values of all the classes in the dataset where the approach can make predictions.

***Precision*$_T$, *Recall*$_T$, *F-score*$_T$:** These metrics are calculated similarly with the above three but are analyzed based on the token level. That is, the method names are split into tokens and the calculation is performed based on the obtained token sets. Note that the token level analysis is widely adopted in studies about method names [15, 58, 69, 86] and the reason can be explained as the quality of a method name prediction depends mainly on the tokens used to compose it [18].

Originally, code2vec and code2seq were assessed on each individual method. To keep consistent with the application scenario of Mario, we re-evaluated them at the class level. Specifically, we calculated their effectiveness based on the prediction results for all the methods in each individual class as introduced above.

Empirical decision for the value of β. To decide the value of β, we conducted a pre-experiment where we set α to 0.5 and changed the value of β from 0.1 to 0.9 with an interval of 0.1. Results showed that the optimal effectiveness was obtained when β equaled to 0.2. Therefore, we empirically set β to 0.2 in our evaluation.

6.4 Results

6.4.1 The effectiveness of Mario *(RQ4).* The performances of Mario and the baselines under five different values of α are shown in Table 1. As we have introduced, the application of Baseline#1 and Baseline#2 does not require a threshold. Therefore, they can make predictions on different class sets compared with Mario (illustrated by the value of $Recall_C$). For instance, Baseline#1 can work for all the classes in the test set (with a $Recall_C$ of 100%), while Mario can make predictions for 87.1% of the classes in the test set when α equals to 0.1. Directly comparing the overall effectiveness each approach achieves on the classes they can work is not completely fair, since the performance of an approach on a certain subset of the classes may be better than its overall performance. For instance, Baseline#1 could perform better on classes with more FR method names since it only targets FR method names. To fairly compare Mario with these two baselines, we also calculated the performances of these two baselines on the set of classes where Mario can make predictions, under different values of α. As shown by the results, Mario outperforms the baseline approaches at both the method name level and token level under all the five values of α. For instance, when α equals to 0.1, the $F\text{-}score_T$ of Mario reaches nearly 45%, while the best performance of the baselines is 25%, which is achieved by Baseline#2. Specifically, the $recall_T$ reaches nearly 55%, indicating that in general more than half of the tokens composing the method names can be predicted by Mario. Similarly, when α equals to 0.9, the $F\text{-}score_T$ of Mario reaches around 65%, while the best performance of the baselines is 55%, which is achieved by Baseline#3. The $F\text{-}score$ values of Baseline#1 on method name level and token level are relatively low (4.0% and 15.2% respectively). This indicates that the current IDEs perform rather poorly on method name prediction for the

Table 1. Comparisons between Mario and the baselines under different values of α (in %).

Approach		$Recall_C$	$precision_M$	$recall_M$	$F-score_M$	$precision_T$	$recall_T$	$F-score_T$
	Baseline#1	100.0	4.7	4.5	4.3	22.2	13.3	15.2
	Baseline#2	94.7	18.0	18.3	16.9	28.0	28.5	24.8
	Code2vec	100.0	27.4	26.2	26.6	43.7	39.6	40.6
	Code2seq	100.0	37.3	36.8	36.9	66.3	63.2	63.4
$\alpha = 0.1$	Baseline#1	87.1	4.4	4.2	4.0	22.3	13.2	15.2
	Baseline#2	87.1	18.0	18.3	16.9	28.0	28.5	24.8
	Baseline#3	87.1	10.1	21.9	11.8	19.7	32.8	20.1
	Mario	87.1	**35.3**	**46.6**	**36.5**	**44.5**	**54.3**	**44.7**
$\alpha = 0.3$	Baseline#1	85.1	4.4	4.2	4.0	22.3	13.2	15.2
	Baseline#2	85.1	18.0	18.3	16.9	28.0	28.7	25.0
	Baseline#3	85.1	16.0	32.0	18.2	24.5	42.0	26.6
	Mario	85.1	**35.8**	**48.2**	**37.2**	**44.4**	**56.3**	**45.4**
$\alpha = 0.5$	Baseline#1	80.6	4.3	4.1	3.9	22.2	13.1	15.1
	Baseline#2	80.6	18.4	19.3	17.4	28.5	29.6	25.4
	Baseline#3	80.6	22.2	40.3	24.5	30.3	49.9	32.7
	Mario	80.6	**34.0**	**42.8**	**34.4**	**44.3**	**53.6**	**44.2**
$\alpha = 0.7$	Baseline#1	68.1	4.1	3.9	3.7	22.1	12.8	14.8
	Baseline#2	68.1	19.3	21.1	18.6	29.6	31.4	26.7
	Baseline#3	68.1	29.9	49.8	33.9	37.7	59.3	42.0
	Mario	68.1	**39.2**	**50.0**	**42.4**	**48.8**	**59.9**	**51.5**
$\alpha = 0.9$	Baseline#1	40.4	3.6	3.4	3.2	21.9	12.3	14.4
	Baseline#2	40.4	22.2	26.6	22.0	32.7	36.3	30.2
	Baseline#3	40.4	48.8	57.5	48.3	56.5	65.0	55.6
	Mario	40.4	**51.9**	**61.8**	**52.5**	**67.1**	**68.9**	**63.6**

class. We conducted the Wilcoxon Signed-Rank Tests [89] to analyze the statistical significance of the difference between the effectiveness of Mario and the baselines. We opted for non-parametric tests since they do not make assumptions about the distribution of the data or the equality of variance and thus provide a more flexible approach to analyzing experimental data [36, 38, 60]. Specifically, we compared the achieved $F\text{-}score_T$ values of Mario on each individual class against those achieved by the three specially-designed baselines. Such a comparison was conducted for the five different values of α and the results show that Mario significantly outperforms the baselines under all the situations. Specifically, all the p-values are less than 0.001 in the comparison results. We also computed the Cliff's delta [31], a non-parametric effect size measure that can evaluate the amount of difference between two variables.[5] Results show that the $F\text{-}score_T$ has at least a medium effect size under all the situations. For instance, when $\alpha = 0.1$, the Cliff's delta values between Mario and the three baselines with respect to $F\text{-}score_T$ are 0.590, 0.418, 0.448, respectively. Such results indicate that the performance differences between Mario and the baselines are significant.

Another interesting phenomenon is that when α varies from 0.1 to 0.5, the performances of Mario only change slightly (e.g., the $F\text{-}score_T$ is always around 45%). Therefore, users of Mario can set three configurations depending on its sensitivity to the potential proximate classes: $\alpha = 0.1$ (high sensitivity); $\alpha = 0.7$ (medium sensitivity); and $\alpha = 0.9$ (low sensitivity). When setting with the *high sensitivity*, Mario can work for nearly 90% of the classes with the $F\text{-}score_M$ and $F\text{-}score_T$ being around 35% and 45%. Although such results may indicate that more manual inspections are required in practice, we note that the $F\text{-}score_T$ of around 45% already exceeds that of the state-of-the-art technique code2vec (i.e., 40.6%). When setting with the *medium sensitivity*, Mario

[5]Cliff defines a delta of less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as negligible, small, medium, large effect size, respectively.

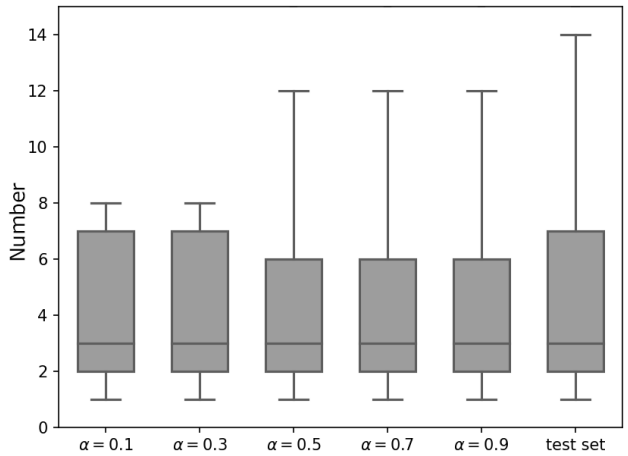


Fig. 6. The distribution of the numbers of method names predicted by Mario on each individual class and the oracle distribution of the numbers of method names in each class from the test set.

Table 2. The performances of Mario on field-relevant (FR) and field-irrelevant (FI) method names (in %).

	$\alpha = 0.1$		$\alpha = 0.3$		$\alpha = 0.5$		$\alpha = 0.7$		$\alpha = 0.9$	
	FR	FI	FR	FI	FR	FI	FR	FI	FR	FI
$precision_M$	57.9	33.7	58.8	34.3	58.9	32.4	58.8	37.9	58.3	52.3
$recall_M$	72.6	44.0	71.9	45.9	71.1	40.3	69.8	48.0	67.6	60.9
$F\text{-}score_M$	59.6	34.3	59.8	35.2	59.5	32.2	58.9	40.8	57.7	52.1
$precision_T$	71.4	41.6	71.9	41.7	72.0	41.6	72.0	46.7	71.8	66.8
$recall_T$	87.0	50.1	86.4	52.5	85.8	49.7	84.9	56.7	83.8	66.9
$F\text{-}score_T$	75.1	40.7	75.2	41.7	75.0	40.6	74.5	48.8	73.9	62.1

can work for nearly 70% of the classes and the $F\text{-}score_M$ and $F\text{-}score_T$ increase to around 45% and 50% respectively. When setting with the *low sensitivity*, which means Mario becomes more strict and can make predictions for fewer classes (i.e., around only 40% of the classes), these results are quite accurate. Especially, the $F\text{-}score_T$ with 63.6% slightly exceeds that of the state-of-the-art technique code2seq (i.e., 63.4%). Furthermore, the $recall_M$ exceeds 60% which means more than half of the oracle method names can be directly predicted by Mario. Such a values also significantly outperforms that of code2seq which is 36.8%.

Since the number of method names to predict is not a fixed value, we therefore also investigated how many method names are predicted by Mario on each individual class and the results are shown in Figure 6. We find that the number of method names predicted by Mario remains consistent to the oracle distribution (i.e., the number of methods in each class from the test set). Specifically, the median values of these distributions are all three. Moreover, with the increase of α, the number of predicted method names does not change a lot. This means that the number of predicted method names has low correlations with the inputs.

The effectiveness of Mario *demonstrates the potential towards the PI-MNP direction. It also outperforms the baselines significantly w.r.t* $F\text{-}score_T$ *under different values of* α.

6.4.2 The effectiveness of Mario *on field-relevant (FR) and field-irrelevant (FI) method names (RQ5).* The effectiveness of Mario on the two types of method names is listed in Table 2. It is clear that the performance of Mario on FR names is significantly better than that on FI names. For instance, when α is 0.1, the $F\text{-}score_T$ on the FR names is 75.1%, nearly twice as that on the FI names (i.e., 40.7%). Also, our statistical test results show that the p-values comparing the $F\text{-}score_T$ values on FR

```
1|  |class DetailActivity {
2|  |
3|✔|  protected void onCreate(){...}
4|✔|  private Intent createShareForecastIntent(){...}
5|✔|  public boolean onCreateOptionsMenu(){...}
6|✔|  public boolean onOptionsItemSelected(){...}
7|  |}
```

(a) the DetailActivity.java from the *ud851-Sunshine* project

```
1|  |
2|  |
3|✔|  onCreate()
4|✔|  createShareForecastIntent()
5|✔|  onCreateOptionsMenu()
6|✔|  onOptionsItemSelected()
7|  |
```

(b) the predicted results from `Mario`

Fig. 7. An example where `Mario` predicts all the method names correctly.

method names and FI method names are all less than 0.001 under the five α values. This is within our expectation as empirical evidence and domain knowledge sometimes perform better than deep learning [53, 59]. However, we also tried to apply the same strategy as we did for Baseline#3 on FI names, that is, we selected the top-8 FI method names (such number is adopted since in our empirical dataset, one class possesses 8 FI method names on average) that occur the most frequently in the proximate classes as the result. The results reveal that this strategy performs even worse. For instance, the $F\text{-}score_T$ is around 20% when α is 0.1. Therefore, how to effectively infer the names for the FI methods deserves further exploration in future.

Another interesting finding is that with the increase of α, the performance of `Mario` on FR names only slightly changes while that on the FI names increases significantly. This indicates that our strategies for inferring FR names, which are based on prior knowledge summarized from Big Code, can generate high-quality prediction results under all thresholds. On the contrary, our deep learning model relies heavily on the input data.

Thanks to the prior knowledge summarized from Big Code, the effectiveness of `Mario` *on the FR names is significantly better than that on the FI names.*

7 CASE ANALYSIS

To better investigate the usefulness of `Mario`, in this section we analyze several cases to showcase how `Mario` could alleviate developers from the burdens of naming methods during development. All the predictions we demonstrate in this section were obtained during our evaluation in RQ4, that is, we pick an existing class from an application, provide its semantic class name to `Mario`, and then check if `Mario` can predict the method names within this class accurately. Note that the predictions are obtained under the *medium sensitivity* (α = 0.7) since `Mario` achieves a promising trade-off between the effectiveness and the generality under such a setting.

Figure 7 shows an example where `Mario` can correctly predict all the method names in the target class, which is from an Android application, *ud851-Sunshine*.[6] The predictions from `Mario` therefore contain common method names in Android applications such as `onCreate`. To our surprise, `Mario` also accurately predicts a non-trivial method name (i.e., `createShareForecastIntent`). This method has attracted much attention from developers as they wrote a detailed Javadoc to describe its functionality ("creating our Forecast intent for sharing"), the operations in the method body ("set the type of content that we are sharing and the text itself"), as well as its return value ("return the newly created Intent"). From this perspective, applying `Mario` could both help developers name the methods accurately and avoid developers missing key functionalities of the class.

[6] https://github.com/udacity/ud851-Sunshine

```
1|  |class WsFrameTextFilter {
2|  |
3|✔|  protected Object doFilterWriteWsBinary(){...}
4|✔|  protected void wsTextReceived(){...}
5|  |
6|  |}
```

(a) the WsFrameTextFilter.java from the *gateway* project

```
1|  |
2|  |
3|✔|  doFilterWriteWsBinary()
4|✔|  wsTextReceived()
5|✗|  ReadFrom()
6|  |
```

(b) the predicted results from `Mario`

Fig. 8. An example where `Mario` predicts an extra method name.

```
1|  |class AttendeeModifiedCommentEmail {
2|  |
3|✔|  public List<String> getHeader(){...}
4|✔|  public String getMessage(){...}
5|✔|  public String getFromAddress(){...}
6|✔|  public String getSubject(){...}
7|✗|  public List<VEvent> generateEvents(){...}
8|  |
9|  |}
```

(a) the AttendeeModifiedCommentEmail.java from the *sakai* project

```
1|  |
2|  |
3|✔|  getHeader()
4|✔|  getMessage()
5|✔|  getFromAddress()
6|✔|  getSubject()
7|  |
8|  |
9|  |
```

(b) the predicted results from `Mario`

Fig. 9. An example where the prediction of `Mario` misses a method name.

Figure 8 shows an example where `Mario` predicts an extra name for the target class. Specifically, besides the two oracle names, `Mario` recommends another name, i.e., `ReadFrom`. This is potentially because the semantic class name of the target class contains a token "text" and `Mario` therefore recommends a common method name from other classes which deal with text information. This example shows that the information of each token in the semantic class name makes sense in the recommendation of `Mario`. Besides, even if extra method names are predicted, developers can still filter the undesired predictions based on their domain knowledge.

Figure 9 illustrates an example where `Mario` successfully predicts most of the method names in the target class but misses a specific one, i.e., `generateEvents`. Here, all the names predicted by `Mario` are field irrelevant because there is no filed named `header` or `message` in the target class. Nonetheless, since such method names are common in email-related classes, `Mario` successfully predicts them. As a comparison, the missed `generateEvents` is more likely to exist in classes which specially serve as event generators to deal with a standard or specified event, such as the linked one.[7] Indeed, in-depth investigation shows that the body of this method only contains a single statement `return null;`, which does not contain much semantic information.

Figure 10 shows an example where `Mario` makes an incorrect prediction for the target class. The target class provides services for users to ask for a leave. It contains a method named `startWorkflow` to start this service and another method named `findTodoTasks` to find out what else should be done before the application will be dealt with. `Mario` successfully predicts the names of these two methods. The target class also contains a method named `complete` to save the tasks performed by the users. However, `Mario` predicts another one named `getDescription`,

[7] https://github.com/gillesdami/QuestPortal/blob/main/app/src/common/shared/org/mozilla/vrbrowser/input/MotionEventGenerator.java

```
1|  |class LeaveWorkflowService {
2|  |
3|✔| public public ProcessInstance startWorkflow(){...}
4|✔| public List<Leave> findTodoTasks(){...}
5|✗| public void complete(){...}
6|  |
7|  |}
```

(a) the LeaveWorkflowService.java from the *activiti-in-action-codes* project

```
1|  |
2|  |
3|✔| startWorkflow()
4|✔| findTodoTasks()
5|✗| getDescription()
6|  |
7|  |
```

(b) the predicted results from Mario

Fig. 10. An example where Mario makes an incorrect prediction.

Table 3. The detailed information of the fifteen classes implemented in the user study.

Project	Class Name[8]	Functionality
Softbot	***.db.Bot.java	Perform some operations in the database
	***.user.UserController.java	Control the users' activities
	***.impl.MarketService.java	Make some service to the softbot market
Online forum	***.controller.LoginController.java	Control the logic of the login activity
	***.controller.UserController.java	Control the users' activities
	***.controller.ManagerController.java	Control the managers' activities
Knowledge base system	***.router.DBRouterJoinPoint.java	Do database routing
	***.aspect.LogAspect.java	Maintain the logs of the system
	***.service.DocService.java	Manage the documents
Risk control system	***.addresssimilarity.addressSimilarityService.java	Calculate the similarity between two IP addresses
	***.dataclean.messageRouteAndSendService.java	Send network messages
	***.dataclean.convertMessageStructureService.java	Convert the data structure of a message
Device update platform	***.dto.ClientLoginDto.java	Transport data for a client login subsystem
	***.impl.IdaasServiceImpl.java	Interact with a third-party system
	***.login.LoginService.java	Fulfill the login activity

which contains weak correlation with the functionalities of the target class. This example suggests that Mario still has spaces for improvements.

8 USER STUDY

To further investigate the practical usefulness of Mario, we performed a user study where we focused on studying how well Mario can help developers in their daily programming tasks.

8.1 Procedure

To perform the user study, we recruited five developers working at top-tier IT companies in China, including Alibaba, Huawei, Tencent, and ByteDance. They all have more than five years of Java development experience and Java is the primary programming language in their daily work. Their current working projects cover a wide range of topics including softbot, online forum, knowledge base system, risk control system, and device upgrade platform.

To perform our experiment, we asked each participant to provide us with the full qualified class names of three classes they were going to implement in their working projects. After that, we showed them the prediction results from Mario and then they implemented the classes with the suggestions from Mario. Table 3 lists the detailed information of these classes as introduced by the participants. As shown in the table, these classes possess diverse functionalities. We confirmed that they had no detailed design about the methods of these fifteen classes before the implementation and the predictions from Mario were the only prompts for them. Finally, after their implementations, they helped us calculate the metric performance of Mario based on their implementations and complete

[8]Due to the confidential policy, we only show the last two words of the full qualified class name.

Table 4. The results of our user study.

	Usefulness	Naturalness	Adequacy	Relevance	Uniqueness
Developer#1	4	5	2	3	4
Developer#2	4	3	2	3	4
Developer#3	5	5	4	4	3
Developer#4	5	4	4	4	5
Developer#5	3	3	2	3	4
Average	4.2	4.0	2.8	3.4	4.0

our online questionnaire which focuses on investigating their feelings about the predictions of `Mario`. Similar to the case analysis in the above section, the predictions of `Mario` are also obtained under the *medium sensitivity* (α = 0.7). Since we have no access to the working projects of the participants, the predictions of `Mario` are only based on the *class repository* without any target project information available. This setting will be discussed later in Section 9.4. All the developers' needs and the predictions from `Mario` are released in our online repository.

In our questionnaire, each participant was asked to rate the prediction results of `Mario` from five aspects: (1) **Usefulness:** reflecting how the predicted names help developers address the method naming problem during their implementations; (2) **Naturalness:** reflecting how the predicted names are similar to those that are used by developers in their daily programming; (3) **Adequacy:** reflecting how the predicted names are abundant compared to the developers' implementations (this aspect assesses the *recall* from developers' perspective); (4) **Relevance:** reflecting how the predicted names are needed by the developers (this aspect assesses the *precision* from developers' perspective); and (5) **Uniqueness:** reflecting how the predicted names are semantically unique to each other (i.e., whether there are duplicate predictions). All scores are integers, ranging from 1 to 5 (1 for poor, 2 for marginal, 3 for acceptable, 4 for good, and 5 for excellent). After rating, the participants could also write additional comments as they like.

8.2 Results

8.2.1 Class Complexity. After implementations, the participants helped us record the lines of code (LOC) and the number of local methods (NLM) of each class, which are two widely-used metrics to assess the class complexity [70, 91]. Results show that on average, the implemented class contains 175.8 lines of code and 8.5 methods. We also investigated that on average, a class from our empirical dataset contains 172.4 lines of code and 8.5 methods. Such results indicate that the implemented classes are similar to the classes from open source projects in terms of the complexity.

8.2.2 Qualitative. The scores given by the participants are listed in Table 4. We note that the developers generally perceive the usefulness of `Mario`: none of them gave a negative score towards the usefulness and the average score for usefulness is 4.2, a relatively high score. Developer#4 leaves a comment: "My merge requests sometimes fail because of the poor method names and re-submissions of the requests take a lot of time, so I find the suggestions very useful". Such results illustrate the rationale of the motivation of this study, i.e., performing the PI-MNP task is useful for developers. We also note that developers give positive feedback towards the naturalness and uniqueness of `Mario`'s predictions. Specifically, both naturalness and uniqueness receive an average score of 4.0. Developer#1 says "I really once used or met the recommended names in my projects or somewhere else, so I feel they are natural".

One weakness of `Mario` identified through the user study is the adequacy of its predictions. The average score towards this point is lower than 3 and three participants gave a marginal score. We recall that a class in our empirical dataset contains around nine methods on average. Therefore, with the $F\text{-}score_M$ being 31.9%, developers may still need to conceive the names for more than six

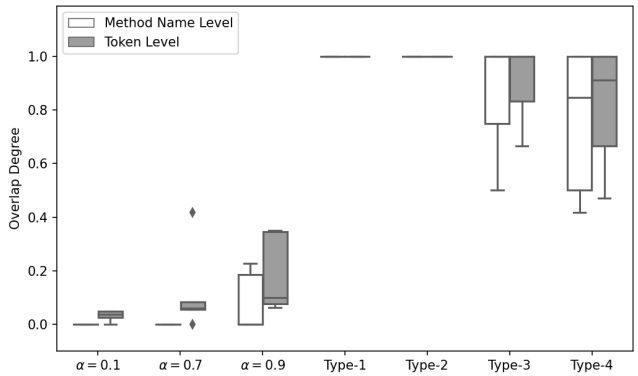


Fig. 11. The overlap degrees of (1) the recommendation results of `Mario` for two classes from the same project and (2) the method names of cloned classes in real-world projects with respect to different clone types.

methods in one class, which might lead to their low ratings towards the adequacy. For instance, Developer#5 leaves a comment: "The predicted method names for the `ClientLoginDto.java` class are not enough". Such results reflect that future studies could focus on enhancing the adequacy of the prediction results when performing the PI-MNP task.

8.2.3 Quantitative. According to the confidential policy, the participants did not share their code with us. Hence, after their implementations, we asked them to help us calculate the performance of `Mario` with respect to the metrics defined in Section 6.3. Results show that on average, the $F\text{-}score_M$ and $F\text{-}score_T$ of `Mario` on the fifteen classes are 31.9% and 44.2%, respectively. Such a performance is similar to that of `Mario` on our test set when the local project information is unavailable and $\alpha = 0.7$ (which will be detailed later in Section 9.4).

9 DISCUSSION

9.1 Will `Mario` Recommend Similar Method Names to Classes from the Same Project and Incur Code Clones?

Given that the application of `Mario` relies on the semantic name of the target class to identify proximate classes, there is thus a concern that `Mario` could generate similar recommendations for classes from the same project since they are likely to possess similar semantic class names and have similar proximate classes. If this is the case, `Mario` will recommend similar method names to classes from the same project and thus may incur code clones. We investigate this potential side effect of `Mario` from two perspectives.

From one perspective, we calculate the similarity between the method names predicted for two different classes by `Mario` and compare such similarities with those of real-world code clone pairs. The intuition behind is that two classes would have similar method names if they are clones, since method names usually represent the key functionalities of a class [7, 8]. Specifically, we randomly selected 1,000 pairs of classes from the test set and compared the recommendation results for them. For each pair of classes, we ensured the two classes (1) are from the same project, and (2) possess the identical rightmost token in their semantic class names (this is because according to our definition, two classes can only have similar proximate classes if the rightmost tokens in their class names are identical). We calculated the Jaccard similarity (which is utilized to represent the overlap degree) between the two sets of recommended names with respect to both method name level and token level, as introduced in Section 4.3. We investigated the results obtained under three different values of α, which correspond to three different configurations (i.e., high sensitivity, medium sensitivity,

and low sensitivity). We also noted that White *et al.* [88] manually checked 398 class-level code clone pairs from eight well-known Java projects, among which 371 pairs were evaluated to be real clones. These clones were found to map to all four clone types (Type-I: identical code, Type-II: identical code except for variations in identifier names and literal values, Type-III: Syntactically similar code but has some statement insertion/deletion/modification, Type-IV: Syntactically dissimilar code that implements the same functionality) and we chose to use these code clones in real-world projects for our investigation. For each clone pair, we also calculated the Jaccard similarity between the method names of two classes. Results are shown in Figure 11. We first note that cloned classes are very similar with respect to their method names. Specifically, our results show that the type-I and type-II class clones have identical method names (the method name level and token level overlap degrees are all 1). For type-III and type-IV class clones, such similarities are also extremely high, with method name level and token level overlap degrees exceeding 0.8. Such results also reveal the rationale of our intuition. In contrast, the overlaps of the recommendations from `Mario` keep in a low degree in general. For instance, under the low sensitivity setting (which demonstrates the highest similarities among the three settings), the median values of the method name level and token level overlap degrees are lower than 0.2. We also performed the one-sided Mann-Whitney U-Test [66] to analyze the statistical significance of the similarity differences with respect to the token level. Our Null hypothesis is that **H0: the token level overlap degree of method names from two cloned classes is not significantly higher than that from the recommendations of `Mario`**, and the Alternative hypothesis is **H1: the token level overlap degree of method names from two cloned classes is significantly higher than that from the recommendations of `Mario`**. Results reveal that the similarity differences are statistically significant (i.e., p-value < 0.01) when comparing the recommendations from `Mario` and the real-world code clones, indicating that **H0** can be rejected with a confidence level of over 0.99. Such results indicate that `Mario` can generally predict different method names for classes from the same project whose rightmost tokens in the semantic class names are identical. Therefore, applying `Mario` in practice would unlikely lead to a proliferation of nearly duplicate classes (i.e., code clones).

From another perspective, whether the code written by developers is a clone mainly depends on the detailed implementation of the class, while `Mario` only recommends the method names and recommends nothing related to the detailed implementations. To demonstrate this point, in our user study, we asked the participants to check if their implemented classes are clones of other classes in the project after they finished their implementations. Note that since we have no access to their working projects, such a check was performed by our participants. We asked them to use the easy-to-deploy and efficient SourcererCC [77], which has been used to analyze large-scale GitHub projects [65], to perform this check. Results show that among the fifteen classes which were implemented under the help of `Mario`, only one was identified to be a clone of other classes. The participant explained this result as "Indeed, I reused some code from other classes during my implementation". Such results reveal that when applying `Mario` in practice, whether a class will be a clone depends on the implementation of the developers, and it is unlikely to incur code clones if the class is appropriately implemented.

9.2 Field Availability

In our evaluation, we assess the overall effectiveness of `Mario` on a class by providing it with all the fields in the class. Nonetheless, applying `Mario` in practice does not require defining all the fields in advance. In fact, `Mario` can predict FR method names each time a field is defined since it only needs proximate classes and the prior knowledge for making such recommendations. Therefore, developers can use `Mario` to recommend a set of FI method names first, possibly with a partially-defined class field set. The developers' knowledge will increase with the implementation

Table 5. The performances of Mario under different values of α when the local project information is unavailable (in %).

	$\alpha = 0.1$	$\alpha = 0.3$	$\alpha = 0.5$	$\alpha = 0.7$	$\alpha = 0.9$
$Recall_C$	63.2	60.5	53.4	37.4	16.1
$precision_M$	22.2	22.8	23.5	27.5	44.2
$recall_M$	35.5	36.1	36.8	41.9	58.0
$F\text{-}score_M$	26.5	27.0	27.7	32.2	48.8
$precision_T$	34.8	34.9	35.9	40.7	58.8
$recall_T$	48.5	49.4	50.7	53.4	65.9
$F\text{-}score_T$	38.9	39.1	40.2	44.4	60.1

of such methods. Finally, with a comparatively complete list of class fields, Mario will return a set of FR method names. Subsequently, in the software iterations, Mario can still work for the newly-defined fields.

9.3 The Complementarity Between Mario and Existing Studies

In this study, we propose Mario, a method name predictor that aims at boosting software development. One thing needs clarification is that Mario and the existing method name recommendation techniques are not mutually exclusive but can be complementary to each other. Mario can propose a skeleton for a class in the early phase of software development while existing techniques can refine method names after detailed implementations have been finished.

9.4 The Effect of the Local Project

In the default setting of Mario, it assumes that the other classes within the same project are available when predicting the method names of a specific class. The rationale for this decision is that due to the bloom of open source, software is not developed from scratch any more [57], which means when implementing a new class, there already exists a number of classes in the project. However, this assumption might not always be true. Therefore, we also assess Mario's effectiveness when the information from the local project cannot be provided. Since the inputs of our model will be changed under such a condition, we re-trained our Transformers for each value of α. The experimental results are shown in Table 5. Specifically, we find that the overall performances of Mario are only slightly affected under different values of α. For instance, compared with the performance obtained when the local project information is available, the $F\text{-}score_T$ drops from 63.6% to 60.1% with a decrease of 3% when α is 0.9. This indicates that without the local information, our utilized "Big Code" can still provide accurate information for prediction, demonstrating the robustness of Mario. Nonetheless, we find that $Recall_C$ experiences significant decreases compared with the values shown in Table 1, especially when $\alpha \geq 0.5$. It suggests that for a large amount of classes, their proximate classes with high similarities only exist in the local project. This phenomenon could be alleviated by enlarging the size of our source code corpora. Hence, in the future, if we include more high-quality repositories, the performance of Mario can be potentially boosted.

9.5 The Effectiveness of Mario when the Information of Local Projects is Partially Available

Our experiment setting in RQ4 can be considered as investigating the effectiveness of Mario when the implementation of a project is nearly finished (because we assume the classes of the local project are all available), while our setting in the above subsection can be considered as investigating the effectiveness of Mario at the beginning of a project (because we assume the classes of the local project are all unavailable). We further investigate the effectiveness of Mario during the

Table 6. The performances of Mario under different values of α when the local project information is partially available (in %).

	$\alpha = 0.1$	$\alpha = 0.3$	$\alpha = 0.5$	$\alpha = 0.7$	$\alpha = 0.9$
$Recall_C$	82.8	80.8	73.6	61.0	34.5
$precision_M$	30.2	32.3	33.1	37.4	47.8
$recall_M$	38.1	44.4	11.6	49.4	61.8
$F\text{-}score_M$	32.6	35.9	36.4	40.8	52.0
$precision_T$	41.4	42.4	42.6	46.4	63.2
$recall_T$	47.2	53.2	52.9	57.7	68.6
$F\text{-}score_T$	42.6	45.2	44.9	48.9	63.1

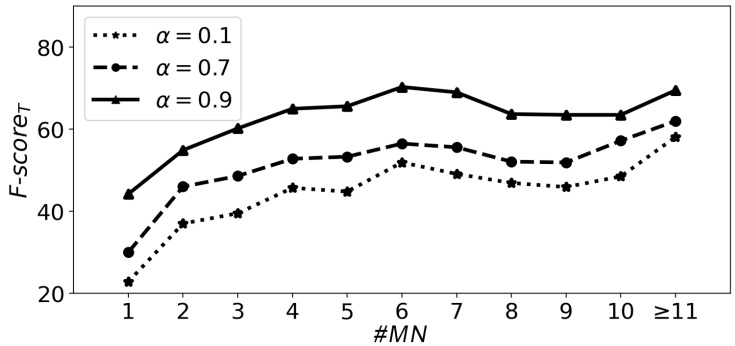


Fig. 12. The performances of Mario on classes with different numbers of method names (#MN) under three sensitivities (in %).

development of a project. Since modern software systems often evolve, which indicates that we could have a number of local classes on hand when applying Mario while some others are going to be implemented in the future. To mimic such an application scenario, we set a time stamp (i.e., 30/6/2017) for the projects in our test set according to their history information. Specifically, classes created before this time stamp are considered as available, and they together with those in the *class repository* are used to make predictions for the classes created after the time stamp. Such a splitting leads to 3,786 classes being evaluated, which account for around 20% of the original test set (3,786/22,822).

Results are listed in Table 6. Note that the $Recall_C$ in this table denotes how many classes out of the 3,786 used ones can Mario make predictions. We note that the achieved $Recall_C$ values are much higher than those achieved when local project information is unavailable (cf. Table 5). This indicates that more information could be utilized by Mario to make recommendations with the evolution of the software. We also notice that Mario demonstrates promising effectiveness under such an experiment setting. Specifically, its $F\text{-}score_T$ achieved when α equals to 0.9 is 63.1%, only slightly lower than the value achieved when all the local project information is available (i.e., 63.6%). Therefore, we conclude that Mario can also work effectively during the incremental development process.

9.6 The Performances of Mario on Classes with Diverse Numbers of Method Names

We also investigated the performances of Mario on classes with different numbers of method names ($\#MN$). The $F\text{-}score_T$ obtained under three settings (α = 0.1, 0.7, 0.9, respectively) are demonstrated in Figure 12. We find that Mario achieves relatively poor performance on classes with small $\#MN$. Especially, when $\#MN = 1$, the values of $F\text{-}score_T$ are 23%, 30%, and 44% under each setting. On the contrary, when $\#MN \geq 6$, Mario achieves comparatively high performances. Specifically, its

$F\text{-}score_T$ can nearly reach 70% when α is 0.9. The behind reason for this phenomenon is that classes with small $\#MN$ rarely possess field-relevant method names. Specifically, when α is 0.9, for the 5,840 classes whose $\#MN \leq 5$, 5,274 (90.3%) of them do not possess any field-relevant method names. Therefore, the performance of `Mario` on such classes is compromised since it cannot effectively deal with FI names as we have mentioned above. We recall that previous studies have shown that the number of class fields are positively correlated with the number of class methods since both of them can indicate the complexity of a class [10, 11]. As a result, when applying `Mario` in practice, developers can expect to obtain more accurate prediction results for classes with more fields.

9.7 The Potential Help from the Requirements

As the first study exploring the PI-MNP direction, we rely merely on the class name to approximate the intended functions of the class. This decision is based on the development experience shared by our contacts at four IT companies. They all expressed that currently, their groups do not have detailed instructions for class functions on hand during the development. Therefore, the class name seems to be the only available information for our target task under most conditions. However, in future work, one would also expect to design a more effective approach for classes where detailed requirement instructions are available.

9.8 Implications

For researchers. Our survey with 101 developers reveals that developers usually name the method first before detailed implementations and they generally consider that recommending the names for the methods to be implemented is useful. This indicates that it has potential to boost developers' daily development activities by performing the proposed PI-MNP task. We are the first to tackle this ambition by proposing to utilize the power of "big data". Both the results of our empirical investigation and evaluation demonstrate the effectiveness of such a strategy: proximate classes are pervasive among real-world projects and can provide strong predictive ability, and the pre-implementation method name prediction based on proximate classes can achieve promising results. This indicates that utilizing the big data for addressing the PI-MNP task is a promising research direction. However, the effectiveness of `Mario` still has a large space for improvement. One weakness exposed through our user study is that the predictions are not considered as adequate by the developers. Future efforts could be devoted to address this concern. For instance, loosening the criteria for the proximate classes may increase the number of proximate classes and improve the adequacy of predictions, but at the cost of sacrificing precision. Also, our quantitative evaluation calls for a more comprehensive class repository to make `Mario` applicable for more classes, which is also a promising direction.

For practitioners. If practitioners are going to integrate `Mario` into their daily activities, we foresee two potential application scenarios from our case studies. First, `Mario` can be directly called after defining a class name and it can predict the names of the methods that are likely to be implemented by the practitioners. Second, since `Mario` sometimes predicts some extra method names (as revealed by our second example in the case studies), it can also be used after the implementation of a class to help the practitioners check if more functionalities are desired. Moreover, given that the predictions of `Mario` may not be adequate, we recommend practitioners to set a low value of α (e.g., 0.1) if possible since `Mario` may predict more method names under such a setting (cf. Figure 6).

9.9 Threats to Validity

Internal threats. In our survey, the participants from IT companies were selected by a contact at each company. The threat of selection bias would always be present when the participants were

not fully randomly sampled. However, given that our survey includes participants from different companies and open-source projects, this threat is thus limited.

Another threat is that the heuristic we used to identify field-relevant method names is based on our observation. There is no study explicitly claiming that a method name related to a field can certainly be split into a verb plus with the field. However, this threat is mitigated as (1) this splitting is consistent with the Java naming conventions as we have introduced in Section 4.1, and (2) we randomly investigated 100 classes within our evaluation dataset and only found one exception where the field-relevant method is named the same as the field.

Our proposed approach builds upon the assumption that in the existing classes in a codebase, the semantics and purpose of a given method are correctly captured by its name. Therefore, the quality of the method names pose threats to our study. Unfortunately, it is impossible to manually check each involved method and the literature approaches always assume that information from top-ranked, well-maintained projects is accurate [16–18, 58, 69, 86]. Nonetheless, this threat is mitigated considering that (1) existing studies have concluded that inconsistent method names (i.e., names that incorrectly reflect the method's functionality) are rather rare in top-ranked real-world projects [64, 86]; and (2) we applied an existing inconsistent method name detection tool [86] on the 100 classes randomly chosen in the above paragraph but detected no inconsistency.

Also, another threat is that whether we can split the method names into tokens accurately. In our study, we use a parser implemented by ourselves which is based on the camel cases and underscore naming conventions (and it works well in some previous studies of our group which need to split identifiers [59–61, 85]). However, its effectiveness can worsen if the names do not strictly follow these conventions. To investigate such a threat, we randomly sampled 100 classes from the test set and manually checked if the splitting results of our parser are consistent with the authors' domain knowledge. We only found one contradiction where the method name `XMLizableBit` is split into `XM`, `Lizable`, and `Bit`, but actually `XMLizable` is the name of a tool. Such results indicate that generally our approach can accurately split method names into tokens and thus the effects of this threat is largely mitigated. Incorporating more advanced identifier splitting technique like LINSEN [32] could further boost the effectiveness of `Mario`, which is left as our future work.

External threats. In this study, we only focus on Java language. This is because many of our definitions and decisions are based on naming conventions of programming languages and Java is the most widely-studied one in this direction. However, the principle of `Mario` is not limited to one specific language. Applying `Mario` to other languages needs to deeply understand their naming conventions and thus is left as our future work.

Our user study only involves the implementations of fifteen classes, which is not a statistically significant number. Therefore, the explanation of our user study should focus more on our qualitative findings rather than the quantitative results.

Construct threats. In our survey, we sent questionnaires to the participants to learn their coding practices without introducing the details of our study (e.g., the `Mario` approach and its potential application scenarios). This is to avoid the bias in the survey results and thus mitigates the threats to the construct validity.

Conclusion threats. In our study, we conclude that our approach significantly outperforms the baselines via a Wilcoxon signed-ranked test. It is possible that the statistically significant relationship is found by chance. However, we believe such a threat is limited since the significance level used in our test is 0.001, a relatively low value.

10 RELATED WORK

10.1 Method Name Recommendation

Given that method names play a critical role in program comprehension, a number of approaches have been proposed to recommend high quality method names. Allamanis *et al.* [12] introduced a language model for source code which can project code token into a vectorial space and then select the name whose resulted embedding is similar to that of the method body. The tokens of program identifiers were used to translate into method names [58, 69, 86]. The program structure information revealed by the Abstract Syntax Tree (AST) is widely utilized in this task. Mou *et al.* [68] proposed a tree-based convolutional neural network (TBCNN), where they integrated a convolutional kernel with programs' ASTs. To increase the learning accuracy, Bui *et al.* [26] fused this model with capsule networks and improved the effectiveness. Code representation techniques that utilize AST paths linking any two leaf nodes in an AST can embed a method body well and then predict its corresponding name. Beyond the tree structure, the graph information which involves *data-flow* and *control-flow* relations also demonstrates its potential. Allamanis *et al.* [15] proposed to represent programs as Program Dependency Graph (PDG) to jointly capture the syntactic and semantic information. Hellendoorn *et al.* [47] combined this model with a sequence model, with the aim to utilize both the semantic structural information provided by the graph and the long-distance information represented by the sequence. All of the aforementioned approaches, however, can only be used after the implementation of the target method. Our study is the first to explore method name prediction without implementation information. Therefore, our study advances the method naming practices by trying to make the names of high quality upon their appearances in the project. Moreover, existing approaches only recommend one method name at a time while our study tries to predict a batch of method names simultaneously.

10.2 Java Language Coding Convention

A wide range of studies focus on the conventions of developers when writing Java programs and their potential applications. Hindle *et al.* [51] provided evidence to the hypothesis that real-world programs are rather repetitive and such a property can be captured well by statistical language models for supporting software development tasks. Tu *et al.* [81] revisited the hypothesis and found that another special property of software is its localness (e.g., some program identifiers only occur in specific files). They thus integrated the statistical language model with a specially designed *cache* component to exploit the localness. Allamanis *et al.* [13] proposed to learn coding conventions such as the identifier naming and formatting from a local codebase and then improve code stylistic consistency. A branch of this field is naming convention, that is, how developers name an identifier during coding. Two widely adopted ways to represent multi-token identifiers are the use of underscores and the use of camel casing [22, 23, 35, 49]. Caprile and Tonella [30] analyzed the grammar of method names and created a set of formal grammar patterns, while Wu and Clause [90] focused on the grammar patterns of test names and designed a heuristic-based approach to automatically detect non-descriptive test names. Abebe *et al.* [6] mined naming conventions for other identifiers in the code. For instance, class names should contain at least one noun and should not contain verbs, while method names should start with a verb. Arnaoudova *et al.* [20, 21] presented the catalogue of linguistic anti-patterns for method and attribute names. They than investigated the presence of linguistic anti-patterns in real-world Java software projects and found they are rather popular. Butler *et al.* [28, 29] mined class naming conventions via investigating the grammar structure patterns of class names and the class name construction patterns related to inheritance. They then investigated the naming conventions of *references* (e.g., fields and local variables) with respect to their constitutions and phrasal structures [28]. A recent study surveyed professional

developers about their opinions towards Java naming conventions derived from the academia [19]. Results reveal that participants very much agree about the importance of various standards. After realizing the general naming conventions, a number of software engineering tasks can be further boosted [7, 8, 27, 37, 42, 50, 79]. For instance, Singer and Kirkham [79] explored the correlation between the class names and the micro patterns [40], which are the implementation patterns within the code. They then designed a prototype tool, checking which classes break coding conventions at code review time. Butler *et al.* [27] introduced a naming convention checking library, Nominal, which looks at abbreviations, phrasal structures, and typography. Gupta *et al.* [42] presented a part-of-speech (PoS) tagger for source code names, which was shown to significantly outperform traditional POS taggers from the NLP field. Abebe and Tonella [7, 8] proposed to extract concepts and relations from the source code via parsing program identifier names and they demonstrated that such a technique can help purify the extracted domain concepts of programs. The above studies build empirical foundations for our study. For instance, inspired by Singer and Kirkham [79] who successfully recommended micro patterns of the class using its name, we also treat the class name as a proxy of programmers' expectations on the functionality of the class.

10.3 Code Completion

Many recommender systems have been proposed to solve a wide range of software engineering tasks [72, 74, 80, 82, 83]. Code completion, which recommends the next contents a developer is likely to type under a specific context, is considered as one of the killer features of current Integrated Development Environments (IDEs) [25, 76]. Originally, the context information [25] and the change history of the project [75, 76] are basis for recommending candidate tokens. Bruch *et al.* [25] recommended candidate tokens according to the current working context. Robbes and Lanza [75, 76] improved code completion via involving the information about how the project is changed recently. Then, a number of statistical language models [46, 63, 81] have been proposed based on the software naturalness [51]. After Hindle *et al.* [51] found that source code contains statistical properties, a number of models are used to perform the code completion task [46, 63, 81], among which N-gram is the most popular one. For instance, Hellendoorn and Devanbu [46] improved the N-gram model by addressing three issues which are unlimited vocabulary, locality, and dynamism. Recently, deep learning techniques [54, 56, 62], adopting a pre-training technique and so on, are also applied. Karampatsis *et al.* [54] proposed an open-vocabulary language for source code which utilizes specially designed algorithms to deal with OoV tokens as well as restrict the size of the vocabulary. Kim *et al.* [56] fed the program's syntactic structure into a transformer model and improved the baseline performance of code completion. Liu *et al.* [62] adopted a pre-trained model to generate contextual embedding for tokens and utilized the predicted type of the next token to help the token prediction. The aforementioned techniques, however, can only recommend one token or at best, the next few tokens for developers. Wen *et al.* [87] supported developers from another perspective, which is predicting the next method they are likely to implement. Their basic idea is to categorize methods into different clusters based on the similarities among different methods determined by a code clone detection tool and then summarize cluster level association rules mined from the evolution process of GitHub projects. Then after obtaining the methods that are already implemented, the proposed approach, FeaRS, selects a suitable association rule and recommends the centroid (i.e., the method with the highest number of edges) of the cluster inferred by the rule. Our `Mario`, predicting the names of all the methods that developers are going to implement in the current class, can also be considered as a code completion recommender. It advances the software development process by saving developers' efforts in designing and implementing a class as well as helping improve software quality (since providing developers with the intentions and functionalities that are not yet implemented is able to improve the quality of the software [39]).

11 CONCLUSION AND FUTURE WORK

Existing approaches for recommending method names assume that the method is already implemented. In practice, however, developers need support in devising names for the methods that are likely to be implemented when a class is created. We introduce this task in the literature as PI-MNP (*Pre-Implementation Method Name Prediction*). We further propose `Mario` for addressing the PI-MNP task. `Mario` takes the class names as inputs and leverages the knowledge from classes having semantically similar names to predict the methods' names for the target classes. Experiment results have demonstrated that `Mario` is effective in predicting method names and offers comparable or higher performance to prior approaches that leverage implementation details.

Our future work will focus on how to utilize the proximate classes for further boosting the software development process. For instance, we foresee a scenario where we can recommend code from proximate classes to developers to help their implementations in the class they are writing.

ACKNOWLEDGMENTS

The authors thank Yibo Wang and Ying Wang from Northeastern University for sharing their experience of performing online survey with developers. The authors also thank Chao Peng from ByteDance, Dengwen Lin from Baidu, Yongfeng Gu from Alibaba, and Pei Zhang from Tecent for helping us disseminate our survey in their groups. The authors further thank Yinyuan Zhang, Xiaojun Wu, Tingting Chen, Mengluan Cai, and Haojun Li for helping us perform the user study.

This work was supported by the National Natural Science Foundation of China No.62002125 and No.61932021, the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001), and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 949014).

REFERENCES

[1] 2022. 15 Java Coding Best Practices. https://xperti.io/blogs/java-coding-best-practices/.
[2] 2022. Eclipse Foundation. https://www.eclipse.org/..
[3] 2022. Java Tutorial - How To Write A Method. https://www.youtube.com/watch?v=qQDGYfQPpGg.
[4] 2022. Stanford CoreNLP. https://github.com/nltk/nltk/wiki/Stanford-CoreNLP-API-in-NLTK.
[5] 2022. Writing New Java Classes. https://www.cs.cmu.edu/~mrmiller/15-110/Handouts/writingClasses.pdf.
[6] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. 2009. Lexicon bad smells in software. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 95–99.
[7] Surafel Lemma Abebe and Paolo Tonella. 2010. Natural language parsing of program element names for concept extraction. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 156–159.
[8] Surafel Lemma Abebe and Paolo Tonella. 2011. Towards the extraction of domain concepts from the identifiers. In *2011 18th Working Conference on Reverse Engineering*. IEEE, 77–86.
[9] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.
[10] Jehad Al Dallal. 2010. Measuring the discriminative power of object-oriented class cohesion metrics. *IEEE Transactions on Software Engineering* 37, 6 (2010), 788–804.
[11] Jehad Al Dallal. 2013. Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology* 55, 11 (2013), 2028–2048.
[12] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 38–49. https://doi.org/10.1145/2786805.2786849
[13] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293. https://doi.org/10.1145/2635868.2635883
[14] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51, 4 (2018), 81:1–81:37. https://doi.org/10.1145/3212695
[15] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net.

[16] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33rd International Conference on Machine Learning*. JMLR.org, 2091–2100.
[17] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net.
[18] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353
[19] Reem S. Alsuhaibani, Christian D. Newman, M. J. Decker, Michael L. Collard, and J. Maletic. 2021. On the Naming of Methods: A Survey of Professional Developers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 587–599.
[20] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering* 21, 1 (2016), 104–158.
[21] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2013. A New Family of Software Anti-patterns: Linguistic Anti-patterns. *2013 17th European Conference on Software Maintenance and Reengineering*, 187–196.
[22] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276.
[23] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To camelcase or under_score. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 158–167.
[24] Dave Binkley, Matthew Hearn, and Dawn Lawrie. 2011. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 203–206.
[25] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*. 213–222.
[26] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*.
[27] Simon Butler, Michel Wermelinger, and Yijun Yu. 2015. Investigating naming convention adherence in Java references. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 41–50.
[28] Simon Butler, Michel Wermelinger, and Yijun Yu. 2015. A survey of the forms of Java reference names. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 196–206.
[29] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Mining java class naming conventions. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 93–102. https://doi.org/10.1109/ICSM.2011.6080776
[30] C Caprile and Paolo Tonella. 1999. Nomen est omen: Analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*. IEEE, 112–122.
[31] Norman Cliff. 1996. Ordinal methods for behavioral data analysis.
[32] Anna Corazza, Sergio Di Martino, and Valerio Maggio. 2012. LINSEN: An efficient approach to split identifiers and expand abbreviations. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 233–242.
[33] Florian Deissenboeck and Markus Pizka. 2006. Concise and consistent naming. *Software Quality Journal* 14, 3 (2006), 261–282.
[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186. https://doi.org/10.18653/v1/n19-1423
[35] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 71–80.
[36] Yuanrui Fan, Xin Xia, Daniel Alencar Da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE transactions on software engineering* 47, 8 (2019), 1559–1586.
[37] Zachary P Fry, David Shepherd, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2008. Analysing source code: looking for useful verb–direct object pairs in all the right places. *IET software* 2, 1 (2008), 27–36.
[38] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Shaomeng Cao, Kechi Zhang, and Zhi Jin. 2023. Interpretation-based Code Summarization. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*.
[39] Malcom Gethers, Trevor Savage, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. CodeTopics: which topic am I coding now?. In *Proceedings of the 33rd International Conference on Software Engineering*. 1034–1036.
[40] Joseph Gil and Itay Maman. 2005. Micro patterns in Java code. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 97–116.

[41] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning word vectors for 157 languages. *arXiv preprint arXiv:1802.06893* (2018).
[42] Samir Gupta, Sana Malik, Lori Pollock, and K Vijay-Shanker. 2013. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 3–12.
[43] Udo Hahn and Inderjeet Mani. 2000. The challenges of automatic summarization. *Computer* 33, 11 (2000), 29–36.
[44] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.
[45] Jingxuan He, Cheng-Chun Lee, Veselin Raychev, and Martin Vechev. 2021. Learning to Find Naming Issues with Big Code and Small Supervision. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 296–311. https://doi.org/10.1145/3453483.3454045
[46] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.
[47] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*. OpenReview.net.
[48] Yoshiki Higo and Shinji Kusumoto. 2012. How often do unintended inconsistencies happen? Deriving modification patterns and detecting overlooked code fragments. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 222–231.
[49] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K Vijay-Shanker. 2014. An empirical study of identifier splitting techniques. *Empirical Software Engineering* 19, 6 (2014), 1754–1780.
[50] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 524–527.
[51] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 837–847. https://doi.org/10.1109/ICSE.2012.6227135
[52] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*. 294–317.
[53] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far are We. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 602–614. https://doi.org/10.1109/ASE.2019.00062
[54] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1073–1085.
[55] Suntae Kim and Dongsun Kim. 2016. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering* 21, 2 (2016), 565–604.
[56] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.
[57] Pavneet Singh Kochhar, Eirini Kalliamvakou, Nachiappan Nagappan, Thomas Zimmermann, and Christian Bird. 2019. Moving from closed to open source: Observations from six transitioned projects to github. *IEEE Transactions on Software Engineering* (2019).
[58] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *Proceedings of the 43rd International Conference on Software Engineering*.
[59] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. Automated Comment Update: How Far are We?. In *The 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 36–46.
[60] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. 2022. Predictive Comment Updating with Heuristics and AST-Path-Based Neural Learning: A Two-Phase Approach. *IEEE Transactions on Software Engineering* (2022), 1–20. https://doi.org/10.1109/TSE.2022.3185458
[61] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-Aware Code Change Embedding for Better Patch Correctness Assessment. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 51 (may 2022), 29 pages. https://doi.org/10.1145/3505247
[62] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.
[63] Fang Liu, Lu Zhang, and Zhi Jin. 2020. Modeling programs hierarchically with stack-augmented LSTM. *Journal of Systems and Software* 164 (2020), 110547.

[64] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 1–12. https://doi.org/10.1109/ICSE.2019.00019

[65] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[66] Henry B Mann and Donald R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[67] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[68] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.

[69] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1372–1384.

[70] Hector M Olague, Letha H Etzkorn, Sherri L Messimer, and Harry S Delugach. 2008. An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study. *Journal of software maintenance and evolution: Research and practice* 20, 3 (2008), 171–197.

[71] Wyatt Olney, Emily Hill, Chris Thurber, and Bezalem Lemma. 2016. Part of speech tagging Java method names. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 483–487.

[72] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. 2017. Supporting software developers with a holistic recommender system. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 94–105.

[73] Václav Rajlich and Norman Wilde. 2002. The role of concepts in program comprehension. In *Proceedings 10th International Workshop on Program Comprehension*. IEEE, 271–278.

[74] Peter C Rigby and Martin P Robillard. 2013. Discovering essential code elements in informal documentation. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 832–841.

[75] Romain Robbes and Michele Lanza. 2007. Characterizing and understanding development sessions. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 155–166.

[76] Romain Robbes and Michele Lanza. 2010. Improving code completion with program history. *Automated Software Engineering* 17, 2 (2010), 181–212.

[77] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168.

[78] CS Saranyamol and L Sindhu. 2014. A survey on automatic text summarization. *International Journal of Computer Science and Information Technologies* 5, 6 (2014), 7889–7893.

[79] Jeremy Singer and Chris Kirkham. 2008. Exploiting the correspondence between micro patterns and class names. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 67–76.

[80] Christoph Treude and Martin P Robillard. 2016. Augmenting api documentation with insights from stack overflow. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 392–403.

[81] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 269–280.

[82] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[83] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (2019), 19:1–19:29. https://doi.org/10.1145/3340544

[84] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 6000–6010.

[85] Shangwen Wang, Kui Liu, Bo Lin, Li Li, Jacques Klein, Xiaoguang Mao, and Tegawendé F Bissyandé. 2021. Beep: Fine-grained fix localization by learning to predict buggy code elements. *arXiv preprint arXiv:2111.07739* (2021).

[86] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. 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)*.

[87] Fengcai Wen, Emad Aghajani, Csaba Nagy, Michele Lanza, and Gabriele Bavota. 2021. Siri, Write the Next Method. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 138–149.

[88] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[89] F. Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[90] Jianwei Wu and James Clause. 2020. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software* (2020).

[91] Yuming Zhou, Baowen Xu, and Hareton Leung. 2010. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software* 83, 4 (2010), 660–674.