



# Recommending Variable Names for Extract Local Variable Refactorings

TAIMING WANG, School of Computer Science & Technology, Beijing Institute of Technology, China

HUI LIU\*, School of Computer Science & Technology, Beijing Institute of Technology, China

YUXIA ZHANG, School of Computer Science & Technology, Beijing Institute of Technology, China

YANJIE JIANG\*, Key Lab of HCST (PKU), MOE; SCS, Peking University, China

*Extract local variable* is one of the most popular refactorings. It is frequently employed to replace occurrences of a complex expression with simple accesses to a newly introduced variable that is initialized by the original complex expression. Consequently, most IDEs and refactoring tools provide automated support for this refactoring, e.g., to suggest names for the newly extracted variables. However, we find approximately 70% of the names recommended by these IDEs are different from what developers manually constructed, adding additional renaming burdens to developers and providing limited assistance. In this paper, we introduce *VarNamer*, an automated approach designed to recommend variable names for *extract local variable* refactorings. Through a large-scale empirical study, we identify key contexts, such as variable initializations and homogeneous variables (variables whose initializations are identical to that of the newly extracted variable), that are useful for composing variable names. Leveraging these insights, we developed a set of heuristic rules through program static analysis techniques, e.g., lexical analysis, syntax analysis, control flow analysis, and data flow analysis, and employ data mining techniques, i.e., FP-growth algorithm, to recommend variable names effectively. Notably, some of our heuristic rules have been successfully integrated into *Eclipse*, where they are now distributed with the latest releases of the IDE. Evaluation of *VarNamer* on a dataset of 27,158 real-world *extract local variable* refactorings in Java applications demonstrates its superiority over state-of-the-art IDEs. Specifically, *VarNamer* significantly increases the chance of exact match by 52.6% compared to *Eclipse* and 40.7% compared to *IntelliJ IDEA*. We also evaluated the proposed approach with real-world extract local variable refactorings conducted in C++ projects, and the results suggest that the approach can achieve comparable performance on programming languages besides Java. It may suggest the generalizability of *VarNamer*. Finally, we designed and conducted a user study to investigate the impact of *VarNamer* on developers' productivity. The results of the user study suggest that our approach can speed up the refactoring by 27.8% and reduce 49.3% edits on the recommended variable names.

CCS Concepts: • Software and its engineering → Software maintenance tools; Integrated and visual development environments.

Additional Key Words and Phrases: Refactoring, Extract Local Variable, Name Recommendation, IDE

\*Corresponding author

---

Authors' addresses: Taiming Wang, School of Computer Science & Technology, Beijing Institute of Technology, Beijing, China, 100081, wangtaiming@bit.edu.cn; Hui Liu, School of Computer Science & Technology, Beijing Institute of Technology, Beijing, China, 100081, liuhui08@bit.edu.cn; Yuxia Zhang, School of Computer Science & Technology, Beijing Institute of Technology, Beijing, China, 100081, yuxiazh@bit.edu.cn; Yanjie Jiang, Key Lab of HCST (PKU), MOE; SCS, Peking University, Beijing, China, yanjiejiang@pku.edu.cn.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s).

ACM 1557-7392/2025/1-ART

<https://doi.org/10.1145/3712191>

## 1 INTRODUCTION

*Extract local variable* is a well-known and widely used refactoring [68, 69]. It involves replacing one or more occurrences of a complex expression with a newly added variable and direct access to the variable. This refactoring simplifies the involved source code and enhances readability and maintainability by providing meaningful variable names for extracted expressions. Consequently, *extract local variable* is frequently employed by developers [38], with studies reporting that it accounts for over 80% of refactorings conducted with automated tool support [69]. This underscores its significance in software development for improving code maintainability and comprehensibility.

Existing IDEs (e.g., *Eclipse* [1], *IntelliJ IDEA* [6], *NetBeans* [9], and *Visual Studio* [10]) offer automated or semi-automated support for *extract local variable* refactoring, typically comprising three main components. The first component validates whether the selected expression can be extracted as a new local variable and determines which occurrences of the expression can be replaced with variable accesses [28]. The second component is to recommend a name for the newly introduced variable, and the third component executes the refactoring by automatically modifying the code based on decisions made in the preceding steps. While IDEs are often accurate in precondition validation and source code modification, variable name recommendation, the second component, tends to be less accurate, with reported accuracy rates of less than 30% in our evaluation. One reason for the inaccuracy of variable name recommendations in IDEs is the emphasis on efficiency without substantial latency. IDEs prioritize fast heuristic-based approaches for variable name recommendation, often at the expense of accuracy. For instance, *Eclipse* employs a series of heuristic rules based on three types of contexts: the initialization itself, the parameters assigned by the initialization, and the data type of the initialization. Another contributing factor is the oversight of crucial contexts by existing IDEs. Our empirical study revealed that IDEs often overlook critical context such as homogeneous variables, which are variables with initializations identical to the expressions being extracted. This lack of context-sensitivity contributes to the low accuracy in variable name recommendations, creating a gap between recommended names and those expected by developers. Inaccurate recommendations not only burden developers but also lead to low-quality variable names that can diminish the readability and maintainability of source code [21, 22]. Therefore, there is a pressing need to improve the accuracy of variable name recommendations in IDEs to enhance developer productivity and maintain code quality.

To address the limitations of existing IDEs in accurately recommending variable names for the extract local variable refactorings, we introduce an automated approach called *VarNamer*. *VarNamer* leverages the surrounding contexts of the refactoring, particularly focusing on the initialization of the variable and the names of its homogeneous variables. Our approach begins with an empirical study aimed at identifying the most informative contexts for constructing variable names. By analyzing a large corpus of code, we determine that the initialization of the variable and the names of its homogeneous variables are particularly valuable in this regard. Based on these findings, *VarNamer* comprises three components: reuse-based name recommendation, generation-based name recommendation, and name selection. The name reuse component leverages the presence of homogeneous variables to suggest suitable names, while the name generation component utilizes the initialization context to generate relevant name candidates. Finally, the name selection component selects the most appropriate name from the generated candidates. To evaluate the performance of *VarNamer*, we constructed a dataset of 27,158 real-world *extract local variable* refactorings mined from the commit histories of 1,000 GitHub repositories. Our evaluation results demonstrate that *VarNamer* significantly outperforms the state-of-the-art IDEs, achieving a 52.6% improvement in exact name matching compared to *Eclipse* and a 40.7% improvement compared to *IntelliJ IDEA*. To assess the extensibility of *VarNamer*, we manually collected a dataset comprising 50 real-world extract local variable refactorings from prominent C++ open-source projects. We then evaluated the C++ version of *VarNamer* on this dataset, and the results indicate that our approach achieves comparable performance across

94 -		95 +	Message message = <b>exchange.getIn();</b>
95 -	SubmitSm template = <b>createSubmitSmTemplate(exchange);</b>	96	SubmitSm template = <b>createSubmitSmTemplate(exchange);</b>
96 -	byte[][] segments = <b>splitBody(exchange.getIn());</b>	97 +	byte[][] segments = <b>splitBody(message);</b>
97		98	
98 -	ESMClass esmClass = <b>exchange.getIn().getHeader(SmppConstants.ESM_CLASS,</b> <b>ESMClass.class);</b>	99 +	ESMClass esmClass = <b>message.getHeader(SmppConstants.ESM_CLASS,</b> <b>ESMClass.class);</b>

Fig. 1. An Example of Extract Local Variable Refactoring

different programming languages. Furthermore, we conducted user experiments where developers utilized our approach to perform the extract local variable refactorings. The results reveal a notable improvement in both time efficiency and edit efficiency. Specifically, our approach facilitated a 27.8% reduction in refactoring time and a 49.3% decrease in the number of edits required for recommended variable names. These findings underscore the practical utility of *VarNamer* in enhancing the developers' productivity and satisfaction with the recommended variable names. Notably, the implementations of the key heuristic rules proposed in this paper have been successfully merged into the mainstream IDE *Eclipse* and are now distributed with its latest releases. The contributions of this paper are as follows:

- We conducted a large-scale empirical study on variable name recommendation for the extract local variable refactorings, marking the first of its kind in this research domain.
- We introduced *VarNamer*, a heuristics-based approach designed to recommend variable names for the extract local variable refactorings. Notably, the key heuristic rules proposed in *VarNamer* have been integrated into mainstream IDE *Eclipse*.
- We curated two datasets comprising real-world extract local variable refactorings, one containing 27,158 instances from Java programs and the other containing 50 instances from C++ programs. These datasets, along with the replication package, have been made publicly available [85].

The rest of this paper is structured as follows. Section 2 presents the definitions of key terminologies in this paper with examples. Section 3 presents how we conducted the empirical study and the corresponding results. Section 4 presents the design details of *VarNamer*, and the evaluation results are presented in Section 5. Section 6 discusses the threats to validity and the limitations of this paper. We review the related work in Section 7, and finally Section 8 concludes this paper.

## 2 BACKGROUND

This section provides an essential foundation for readers to establish a common understanding of the concepts discussed throughout the paper. In this section, we introduce five key terminologies by explaining concepts and giving definitions alongside illustrative examples.

**Extract local variable refactoring** (also known as *introduce local variable refactoring* [48]) is a widely used refactoring technique that replaces expressions with a newly introduced local variable and its references. When expressions become complex and difficult to interpret, it is better to extract them as variables with meaningful names to enhance their self-explanatory nature. After extraction, multiple occurrences of the expression can be replaced with references to the new variable. Another benefit of this refactoring is the removal of duplicate expressions that appear multiple times within a single method, thereby reducing the complexity of the enclosing method and ensuring the expression is executed only once. This benefit is particularly desirable when the repeated expression is lengthy or resource-consuming (e.g., invoking a computation-intensive method). A typical example of extract local variable refactoring is shown in Fig. 1. Here, the expression “*exchange.getIn()*” [20] (highlighted in yellow) appears twice within a single method (lines 96 and 98 on the left). Conducting this refactoring, replacing the expression with a local variable named “*message*” in lines 97 and 99 on the right, reduces the complexity of the enclosing method and improves its brevity.

```

1 // Method where the extract local variable refactoring happened.
2 default String packageName() {
3 -     return name().substring(0, name().lastIndexOf('.'));
4 +     final int dotIdx = name().lastIndexOf('.');
5 +     if (dotIdx < 0) {
6 +         return "";
7 +     } else {
8 +         return name().substring(0, dotIdx);
9 +     }
10 }
11 // Sibling method where the homogeneous variable is retrieved
12 default String simpleName() {
13     // HomogeneousVariable: dotIdx
14     final int dotIdx = name().lastIndexOf('.');
15     if (dotIdx < 0) {
16         return name();
17     } else {
18         return name().substring(dotIdx + 1);
19     }
20 }

```

Listing 1. Example of Homogeneous Variable.

**Initialization** of a local variable refers to the expression used to initialize the variable. During an extract local variable refactoring, developers typically introduce a local variable and initialize it with the extracted expression. A typical example is illustrated in Fig. 1. In this example, the *initialization* is “`exchange.getIn()`” (highlighted in yellow) in lines 96 and 98. Notably, the *initialization* serves as the most direct context for recommending the variable name. During our empirical study and recommendation process, we only consider the identifier tokens of the initializations, such as “`exchange getIn`”.

**Data type** of an *initialization* refers to the data type of the newly introduced variable after the refactoring. During an extract local variable refactoring, developers sometimes use the data type as the name of the newly introduced variable. A typical example is “`Message`” in line 95 of Fig. 1. Similar to the preprocessing of the *initialization*, we only retain the identifier tokens of the *data type*.

**Assignment** represents the relationship where parameters, variables, or fields are assigned by an *initialization*. According to an empirical study conducted by Liu et al. [61], there can be high lexical similarity between method arguments and parameters. In such cases, the names of these assigned objects can be considered as part of the variable names. For example, in line 96 of Fig. 1, the *initialization* “`exchange.getIn()`” is assigned as an argument to the method “`splitBody`” whose complete signature is “`protected byte[][] splitBody(Message message)`”. Here, the *assignment* corresponds to the formal parameter name of the method “`splitBody`”, which is “`message`”.

**Declaration Context** represents the **data type** and **initialization** within the variable declaration statement (i.e., except for the variable name itself). This is the most intuitive context for variable name recommendation. Thus most of the popular IDEs, e.g., Eclipse and IDEA, have developed heuristic rules to recommend variable names according to such declaration contexts.

**DEFINITION 2.1. Homogeneous Variable:** *A variable that shares the same initialization expression as the initialization intended for extraction. These homogeneous variables are crucial contexts for recommending names during the extract local variable refactoring process. However, they are often overlooked by the current implementations of popular IDEs, e.g., Eclipse and IDEA. Homogeneous variables can exist within various scopes, including the same Java file where the refactoring occurs, in Java files within the same package, or in Java files within the same project.*

A typical example of *homogeneous variable* is presented in Listing 1. The first method in this example, named “*packageName*”, is where the refactoring occurred, while the second method, named “*simpleName*”, is a sibling method in the same Java file as “*packageName*”. The *Initialization*, highlighted in bold font (lines 3 and 4), is “*name().lastIndexOf(?)*”. The ground truth name, provided by the original developer, is “*dotIdx*” (line 4). Interestingly, at the time of the refactoring, a variable named “*dotIdx*” (line 14) already exists in “*simpleName*” and is initialized by the *Initialization*. Since developers have already assigned a name to the to-be-extracted *Initialization*, we can reuse this name for the recommendation. This variable, due to its high similarity with the newly extracted variable, is referred to as a *homogeneous variable*.

### 3 EMPIRICAL STUDY

In this section, we present the methodologies and findings of the empirical study conducted to explore the most influential factors in variable name recommendation for the extract local variable refactorings. Through this study, we aim to identify and analyze the contexts that contribute most significantly to the effectiveness of variable name recommendations. It is worth noting that we do not discriminate *token* and *sub-token* in this paper, and they both refer to a word in identifiers, e.g., “*simple*” in “*simpleName*”.

#### 3.1 Research Questions

The empirical study should answer the following questions:

- **RQ1:** How often could the name tokens of the extracted variables be found in the contexts of the variables?
  - **RQ1-1:** What are the possible contexts of the variables where the tokens of variable names can be found?
  - **RQ1-2:** How often could the name tokens be found in different types of contexts?
- **RQ2:** How accurate is it to recommend variable names by simply copying (reusing) the names of homogeneous variables?

RQ1 concerns where we can retrieve the tokens to compose the complete name for the newly extracted variable, and how often we can find them in different types of contexts. We further split RQ1 into two sub-questions, i.e., RQ1-1 and RQ1-2. To address RQ1-1, we conducted a sample analysis to identify the most influential factors (i.e., context) contributing to variable name recommendation. The investigation results suggest that there are four types of essential contexts. Consequently, we extended our analysis to a larger dataset and examined how often can the name tokens be found in these four types of context (RQ1-2). The investigation results suggest that homogeneous variables are critical and they often contain the desired name tokens. Inspired by such a finding, we investigated RQ2 with a detailed analysis of the challenges associated with copying names from homogeneous variables, providing insights for the development of heuristic rules in leveraging this context effectively. Answering RQ1 and RQ2 would significantly facilitate the design of context-based approaches to recommending variable names for the extract local variable refactorings.

#### 3.2 Methodology

**3.2.1 Data Collection.** To conduct the empirical study, we first constructed a dataset by collecting real-world extract local variable refactorings from open-source projects on GitHub [4]. We selected the top 1,000 open-source Java projects on GitHub sorted by stars. This selection criteria of projects was made to ensure a diverse and representative sample of real-world refactorings. From each of the selected projects, we collected the extract local variable refactoring as follows:

- First, we leveraged RefactoringMiner [78] to discover a list of *potential* extract local variable refactorings (noted as *pRs*) that have been conducted in the given project. We selected RefactoringMiner because it represents the state-of-the-art method in the automated discovery of refactoring histories [77].

```

408 -         if (withSelf) reconcileWithSelf(seed, rng, rngCallSites,
allocations, builder);
414 +             RecordOption withRng = RecordOption.values()[rng + 1];
415 +             RecordOption withTime = RecordOption.values()[time + 1];
416 +             if (withSelf) reconcileWithSelf(seed, withRng, withTime,
allocations, builder);
417 -             else if (allocations) throw new IllegalArgumentException("--with-
allocations is only compatible with --with-self");
418 +             else if (allocations) throw new IllegalArgumentException("--with-
allocations is only compatible with --with-self");
419 -             else reconcileWith(dir, seed, rng, rngCallSites, builder);
420 +             else reconcileWith(dir, seed, withRng, withTime, builder);

```

Fig. 2. An Example of A False Positive

- Second, we filtered out false positives in *pRs* automatically with static analysis. A *potential extract variable refactoring* was deemed a false positive if the extracted expression did not appear in the original version of the source code (i.e., before refactoring) or if it was not replaced with the newly introduced variable in the new version (i.e., after refactoring). An example is presented in Fig. 2. The *initialization* (highlighted in orange) was “*RecordOption.values()[rng+1]*” [19]. RefactoringMiner reported it as an extract local variable refactoring because a new local variable “*withRng*” was created, and it was used in the original expression (line 416 on the right and line 408 on the left). However, we noticed that this expression did not exist before the refactoring was conducted (i.e., line 408 on the left does not contain this expression). The modification was more like an addition of new variables (lines 414 and 415 on the right) and bug fixing with the new variables (lines 416 and 418 on the right) than an extract local variable refactoring.
- Finally, we removed duplicate refactorings caused by branch merge. If a refactoring was conducted on one commit that was located in branch A, the refactoring would be recorded by another commit where branch A was merged with the main branch. As a result, the same refactoring would be discovered twice, resulting in duplicate refactorings.

We finally obtained 32,039 real-world extract local variable refactorings from 745 projects. Such refactorings were further divided into two disjointed datasets: *EmpiricalDataSet* and *TestingDataSet*. The former was composed of 4,881 extract local variable refactorings discovered from randomly selected 100 projects. It was used for the empirical study in this section. *TestingDataSet* was composed of the other refactorings (27,158 refactorings in 645 projects) and was employed for evaluation in Section 5.

**3.2.2 Investigation of RQ1.** To answer RQ1-1, we randomly sampled 50 extract local variable refactorings from *EmpiricalDataSet*, and analyzed each of them as follows:

- We leveraged spacy [46] to split the name of the extracted variable into sub-tokens.
- For the sub-tokens in the variable name, we located all their occurrences within the Java file where the refactoring happened.
- For each occurrence of the sub-token, two authors independently identified the type of the context (e.g., variable initialization, the data type of the variable, and homogeneous variables) that contained the sub-token. The participants were Ph.D. students majoring in computer science, with more than 4 years of Java programming experience and more than 3 years of research experience in software analysis. The Cohen’s kappa coefficient [30] is 0.87, indicating a strong agreement between the two raters. Any discrepancies were resolved through discussion until a consensus was reached.

With the sample analysis introduced in the preceding paragraph, we can identify the most critical contexts for variable name recommendation. As indicated by the results in Section 3.3, these essential contexts include the initialization of the extracted variable (referred to as *initialization*), *homogeneous variables*, the *data type* of the extracted variable, and *assignments*. Understanding and leveraging these contexts are fundamental for developing effective strategies for recommending variable names in extract local variable refactorings. To investigate RQ1-2, we further validated the findings inferred from the small sample set (50 refactorings) and examined the frequency

of name tokens of variables appearing in the four types of contexts across the entire *EmpiricalDataSet*. Note that *EmpiricalDataSet* comprised 4,881 refactorings. The validation process proceeded as follows:

- First, we obtained variable names and their corresponding contexts, including *initialization*, *homogeneous variables*, *data type*, and *assignments*, through static code analysis using *Eclipse JDT* [3]. This resulted in a list of 5-tuples: [Variable Name, Initialization, Homogeneous Variable, Data Type, Assignment]. Note that we parsed the resulting initialization and data type to remove the operators and separators, e.g., “.”, “(”, and “)” leveraging javalang [7] because such tokens would not appear in variable names.
- Second, we used spacy [46] to split the items in the 5-tuple into sub-tokens. Subsequently, all sub-tokens were converted to lowercase, consistent with previous name recommendation studies [55, 63, 70].
- Third, by comparing the sub-tokens in a given variable name against sub-tokens in its contexts, for each type of context we obtained a list whose size is the number of the sub-tokens in the variable name. For the example presented in listing 1, the variable name (i.e., “dotIdx”) had 2 sub-tokens (“dot” and “idx”). The corresponding result list is [[0,0],[1,1],[0,0],[0,0]], where “1” represented that the sub-token could be found in the corresponding context, and “0” otherwise. It is worth noting that for a single refactoring, there is only one *initializations* and one *data type*. However, it may have multiple *homogeneous variables* and multiple *assignments*. In these cases, we concatenated all names in *homogeneous variables* (or *assignments*) with a blank space, e.g., {*dot idx dot idx dot idx*}.
- Fourth, we further investigated how often the to-be-recommended variable names were identical to the contexts. Unlike the above token analysis, we did not perform the splitting, and thus each unit is a variable name instead of one sub-token. We employ a methodology similar to the above analysis, which we omit here to avoid repetition.

**3.2.3 Investigation of RQ2.** To answer RQ2, we conducted the empirical study as follows. First, we investigated how the distance between the newly extracted variable and its homogeneous variables influenced the performance of recommendations. We categorized distances into three levels: within the enclosing project, within the enclosing package, and within the enclosing document, i.e., the Java file. Subsequently, we quantitatively assessed the impact of distance-based filtering on the performance of recommendations. Second, we observed that not all retrieved homogeneous variable names perfectly matched those of the newly extracted variables. To determine when it is appropriate to reuse the names of homogeneous variables and to explore how we can strategically select the most promising homogeneous variables for variable name recommendation, we randomly selected 100 refactorings from our dataset, comprising 50 successful and 50 failed cases for qualitative analysis. Successful cases refer to instances where the names of homogeneous variables are identical to the variable names. In this case, reusing these names will result in a successful recommendation. Conversely, failed cases occur when the names of homogeneous variables differ from the variable names, indicating that reusing these names will not lead to a successful recommendation. Following a standard brainstorming methodology, we derived four special cases that need to be carefully addressed. To ensure the reliability of these results, we repeated this task on a new dataset. To be specific, we collected 20 additional Java open-source projects that were not included in the original set of 1000 projects. Using RefactoringMiner [78], we identified the extract variable refactorings from their commit history. We then filtered out the invalid refactorings using the same method as in our previous data construction process (Section 3.2.1), resulting in 5,374 valid refactorings. From such refactorings, we sampled another 100 refactorings, consisting of 50 successful cases and 50 failed cases, to repeat the task. As a result, we still identified the four special cases illustrated in Section 3.4.2. By refining the study design and analyzing both quantitative and qualitative aspects, we gained insights into the factors influencing the effectiveness of variable name recommendations based on homogeneous variables.

Table 1. Useful Contexts for Variable Name Recommendation

	Initialization	Homogeneous Variable	Data Type	Assignment	Declaration Context	All Context
# Hitting	4,720	1,155	2,229	81	5,146	5,592
Hitting Rate(%)	57.3	14.0	27.1	1.0	62.5	68.0

Table 2. Chance of Exact Match

	Initialization	Homogeneous Variable	Data Type	Assignment
#Exact Match	17	603	674	21
Chance of Exact Match (%)	0.3	12.4	13.8	0.4

### 3.3 RQ1: Useful Contexts for Variable Name Recommendation

3.3.1 *RQ1-1: Useful Contexts.* Based on the manual analysis of 50 real-world extract local variable refactorings as introduced in Section 3.2.2, we have identified four useful contexts (refer to Section 2 for details) that in total contain 84 sub-tokens composing the names of the extracted local variables.

With the examples presented in Fig. 1 and Listing 1, we illustrate how we identified possible contexts containing the name tokens of variables. We first examined the most intuitive and direct context: *declaration context*, i.e., *initialization* plus *data type*, for name recommendation. In the two examples above, the sub-tokens of the expected variable names are {"message"} and {"dot", "Idx"}. Although neither the *initialization* contains any sub-tokens, the *data type* in the first example is "Message", which is identical (ignoring case) to the expected variable name.

Intuitively, the extracted variable should fit into the statement from which it was extracted. Consequently, besides the *declaration context*, we also examined the statement where the *initialization* is extracted. We found that the expected variable name sometimes matches the parameter name (i.e., *assignments*) if it is assigned by the *initialization*. In the example presented in Fig. 1, "exchange.getIn()" is assigned as an argument to the method "splitBody", and the corresponding parameter name is "message", which is identical to the expected variable name.

We then extended our search scope to other parts of the enclosing method and even the entire Java file. We found that sometimes there are variables with the same name initialized by *initialization* elsewhere in the Java file, which we refer to as *homogeneous variables*. An example is shown in Listing 1, where the *homogeneous variable* found in method "simpleName" contains all the sub-tokens of the expected variable name, i.e., {"dot", "Idx"}.

Following the above procedures, we identified all the name tokens in the above four contexts. Specifically, 61.9% of the name tokens could be found in the *initialization* context, 13.1% in the names of *homogeneous variables*, 27.4% in the *data type* context, and 2.4% in the *assignment* context. It's worth noting that a single sub-token could simultaneously appear in different contexts.

*Answer to RQ1-1:* The most useful contexts for variable name recommendation include *declaration context* (i.e., *initialization* plus *data type*), *homogeneous variable*, and *assignment*.

3.3.2 *RQ1-2: Frequency of Name Tokens in Different Context.* We then extended our analysis across the *EmpiricalDataSet*, and the results are presented in Table 1. The first line denotes different types of context. The second line denotes the hitting number, i.e., the number of sub-tokens that can be found in this context. The last line denotes the

hitting rate which is calculated by the hitting number divided by the total number of the sub-tokens in all the variable names in *EmpiricalDataSet* (8,231). From Table 1, we observe that:

- We have a great chance (68.0%) to find sub-tokens for the to-be-named variable from the given four categories of contexts.
- The declaration contexts present a significant opportunity, with a 62.5% likelihood of containing sub-tokens of the variable name. However, by including additional contexts such as *homogeneous variables* and *assignments*, the likelihood can be substantially improved by  $8.8\% = (68.0\% - 62.5\%) / 62.5\%$ .
- *Homogeneous variables* and *data types* have considerable chances (14.0% and 27.1%) to contain sub-tokens of the variable name.

Overall, the above findings underscore the importance of considering additional contexts, such as *homogeneous variables*, in variable name recommendations for the extract local variable refactorings.

We also examined the frequency of exact matches between the to-be-recommended variable names and their respective contexts. The results are summarized in Table 2. Here, *#Exact Match* represents the number of refactorings where the variable names perfectly match the corresponding context. The *chances of Exact Match* is calculated by dividing the *#Exact Match* by the size of *EmpiricalDataSet* (4,881). From Table 2 we observe that:

- *Homogeneous variables* and *data types* exhibit the highest likelihoods (12.4% and 13.8%, respectively) of being identical to the variable name. This is primarily because *homogeneous variables* and *data types* often share names with the variables themselves, suggesting that recommending variable names by reusing those of *homogeneous variables* and *data types* could be straightforward and accurate.
- Despite over half (57.3%) of the sub-tokens of variable names being found in the *initializations* (as shown in Table 1), the rate of variable names identical to *initializations* is much lower (0.3%). This suggests the complexity involved in extracting tokens from *initializations*.
- *Assignments* exhibit a low rate of exact match (0.4%), consistent with its performance of sub-token occurrences (1.0%).

In conclusion, our study sheds light on the complexities and opportunities in variable name recommendation for the extract local variable refactorings. While *initializations* often harbor a significant number of sub-tokens, the process of extracting them accurately can pose challenges. On the other hand, *homogeneous variables*, despite containing fewer sub-tokens, offer a straightforward and reliable resource for variable name recommendation. This suggests that leveraging *homogeneous variables* may present a simpler and more accurate approach compared to relying solely on *initializations*.

*Answer to RQ1-2:* The name tokens of variables appear most frequently in *declaration contexts*, with *initializations* being the most common, followed by *data types*. *Homogeneous variables* also play a significant role, while *assignments* contribute the least. However, in terms of the likelihood of exact matches, *homogeneous variables* and *data types* exhibit the highest probabilities.

### 3.4 RQ2: Reuse Names of Homogeneous Variables

**3.4.1 Searching Scope.** In Table 3, we present statistics regarding the search for homogeneous variables across different scopes, as introduced in Section 3.2.3. The first row delineates three scopes: the same project, the same package, and the same document, i.e., Java files, from which homogeneous variables can be retrieved. The second row, i.e., “#fruitful cases”, indicates the number of cases where at least one homogeneous variable can be found within the specified distance. The third row, i.e., “#correct cases”, denotes the number of cases where correct variable names can be selected from all homogeneous variables. The fourth row, i.e.,  $P_{correct}$ , denotes the

Table 3. Homogeneous Variables in Different Scopes

Metrics	Project	Package	Document
# Fruitful Cases	1,822	1,326	879
# Correct Cases	1,316	1,002	701
$P_{correct} (\%)$	72.2	75.6	79.7
Avg. # Homogeneous Variable	83,960.6	1,209.7	2.6
Avg. Time Cost (ms)	2,250.8	116.1	0.6

probability of selecting the correct variable name among all homogeneous variables, i.e.,

$$P_{correct} = \frac{\# \text{ correct cases}}{\# \text{ fruitful cases}} \quad (1)$$

The fifth row illustrates the average number of retrieved homogeneous variables, while the sixth row depicts the average time cost of searching for homogeneous variables within the given search scope.

From Table 3, we make the following observations:

- Firstly, as the scope narrows down from the enclosing project to the enclosing package/document, the number of fruitful cases decreases from 1,822 to 1,326 and 879, respectively. However, the possibility of selecting the correct variable name among all homogeneous variables improves from 72.2% to 75.6% and 79.7%. This phenomenon occurs because a closer distance between the recommended variable and its homogeneous variables increases the likelihood of them playing similar roles in their context, thereby sharing the same name.
- Secondly, in terms of efficiency, narrowing down the scope notably reduces the overwhelming number of homogeneous variables (83,960.6 vs. 1,209.7 and 2.6) that require further exclusion and identification. Consequently, the time cost significantly decreases (2,250.8ms vs. 116.1ms and 0.6ms).
- It's worth noting that the involved projects in *EmpiricalDataSet* are large, with an average of 2,701 Java files per project. Consequently, searching all files within the enclosing project for homogeneous variables can be time-consuming. Conversely, confining the search within a single file can significantly reduce the computational cost.

In conclusion, by narrowing down the scope to the same project, package, or Java file, we observe a trade-off between the number of fruitful cases and the efficiency of the selection process. Given our objective of integrating our approach into mainstream IDEs, we have decided to limit the search scope to the same Java file during the evaluation. This choice is motivated by the desire to optimize the efficiency of our approach within the context of typical development workflows. By focusing on the same Java file, we aim to minimize computational overhead while still providing meaningful and accurate variable name recommendations directly within the developer's immediate coding environment.

**3.4.2 Special Cases.** As outlined in Section 3.2.3, we have conducted a qualitative analysis to refine the accuracy of reusing the names of *homogeneous variables*. This analysis has yielded valuable insights into scenarios where it is appropriate or inappropriate to reuse such names, contributing to the enhancement of our variable name recommendation approach. Here are the four typical cases we found:

**Case1** When an *initialization* is a *universal initialization*, reusing the names of *homogeneous variables* is not advisable. *Universal initializations* are those that have been used to initialize different variables (with different names) and have appeared extensively across multiple projects. For instance, “*null*” serves as a *universal initialization* since it is prevalent in initializing any newly created instances. Similarly, “*0*” and

“1” are prevalent in initializing any Integer objects. In addition, “true” and “false” are standard choices to initialize any Boolean objects. Finally, using “new StringBuilder()” is standard for creating an instance of “StringBuilder” across different projects, although the variable names may vary. In such cases, reusing names from *homogeneous variables* may lead to inaccuracies, as these variables may not necessarily resemble the newly extracted variable.

- Case2** If the *initialization* is long enough in character length, it is advisable to reuse their names. Longer initializations are typically more complex and specific in functionality, reducing the likelihood of finding homogeneous variables with similar initializations. Therefore, if any such rare and specific homogeneous variables are found, it is highly reliable to reuse their names.
- Case3** When the *homogeneous variable* and the variable to be extracted play similar roles within the same statement-level context, reusing their names is advisable. The statement-level context refers to the parent statement where the variables are referenced. For instance, in the example provided in Listing 1, both the to-be-extracted variable and its *homogeneous variable* are referenced within statements such as “return name().substring(0, dotIdx);” (line 8) and “return name().substring(dotIdx + 1);” (line 18), exhibiting structural and literal similarity. Therefore, reusing the name of such a *homogeneous variable* can achieve high accuracy.
- Case4** When the *homogeneous variable* and the variable to be extracted serve similar purposes within the method-level context, reusing their name is advisable. For instance, in the example provided in Listing 1, the bodies of the enclosing methods, namely “packageName” and “simpleName”, exhibit significant structural and literal similarity. This similarity in the method-level context suggests that the *homogeneous variable* can serve as a reliable reference, and reusing its name may yield high accuracy.

*Answer to RQ2:* Simply copying (reusing) the names of homogeneous variables has great potential for recommending correct variable names. The possibility of finding a correct variable name among homogeneous variable names (i.e.,  $P_{correct}$ ) increases as the search scope narrows down. Simply copying names from homogeneous variables has a success rate ranging from 72.2% to 79.7%.

## 4 APPROACH

### 4.1 Overview

The overview of *VarNamer* is depicted in Fig. 3. It comprises three key components: rule-based variable name recommendation, generation-based variable name recommendation, and the final name selection and recommendation. For brevity, these components are referred to as name reuse, name generation, and name selection throughout the paper. Initially, *VarNamer* searches for *homogeneous variables* within the enclosing Java file. Upon retrieval, it applies filtering and validation techniques to identify reliable candidates. Simultaneously, *VarNamer* employs conventional naming rules extracted from high-quality code corpora to generate potential names. Finally, a series of heuristic rules are leveraged by *VarNamer* to select the recommended name for developers. Detailed implementation specifics will be discussed in subsequent sections.

### 4.2 Reuse-Based Variable Name Recommendation

Through our investigation of real-world data, we have observed that blindly reusing the names of homogeneous variables in all cases may not always yield satisfactory results. To maximize the utility of homogeneous variables, we have devised a two-pronged approach consisting of a coarse-grained filter and a fine-grained validator, addressing common scenarios identified in Section 3.4.2.

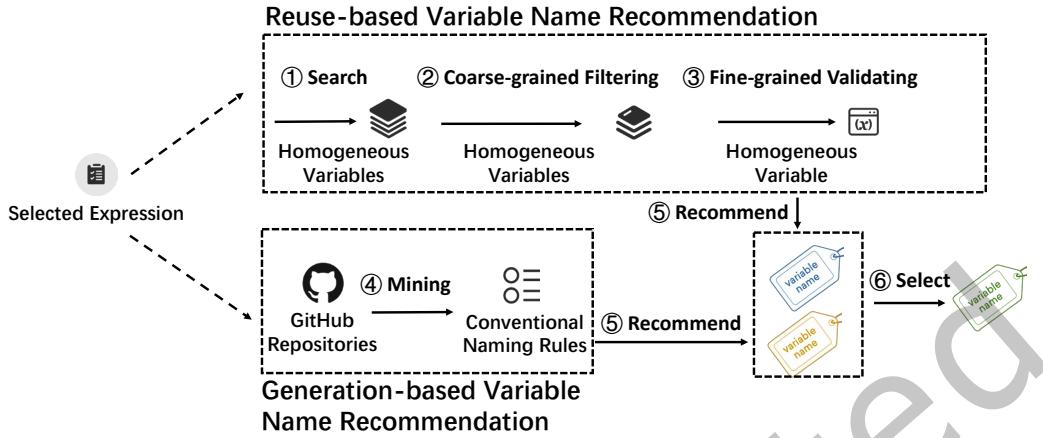


Fig. 3. Overview of VarNamer

**4.2.1 Coarse-grained Filter.** The coarse-grained filter is tailored to tackle **Case1** outlined in Section 3.4.2. We identify *universal initializations* through the following steps: First, we gather initializations assigned with two or more distinct names across each project in *MiningDataSet* (refer to Section 4.3.1). This results in a set of *project-specific universal initializations*. Subsequently, we deem a *project-specific universal initialization* as a *universal initialization* if it appears extensively across multiple projects. The parameter *ProjectNum*, denoting the number of projects, is configurable (refer to Section 5.5). The coarse-grained filter operates as follows: it initially verifies whether the initialization qualifies as one of the *universal initializations*. If it does, the name reuse component is deactivated, and the name generation component is utilized instead. Conversely, if the initialization does not meet the criteria for *universal initializations*, it undergoes further validation by the subsequent fine-grained validator to determine whether to reuse the names of *homogeneous variables*.

**4.2.2 Fine-grained Validator.** The fine-grained validator is devised to handle **Case2**, **Case3**, and **Case4**, as delineated in Section 3.4.2. To identify **Case2**, we compute the character length of the *initializations* and establish a tunable parameter *IniLength* (refer to Section 5.5) to distinguish reliable homogeneous variables from unreliable ones. If the character length of the *initialization* exceeds *IniLength*, it is deemed a reliable homogeneous variable; otherwise, further validation is performed to assess if it falls under **Case3** or **Case4**. For **Case3**, we measure the similarity of the statement-level context (i.e., parent statement where the variables are referenced) between the to-be-extracted variable and the *homogeneous variable*. If the similarity surpasses a tunable parameter *FGSim*, it is considered reliable. Regarding **Case4**, we assess the similarity between the enclosing method of the homogeneous variable and that of the to-be-extracted variable. If multiple homogeneous variables persist after the aforementioned filtering and validation, only the one with the highest method similarity is retained. The measurement of these two similarities is outlined as follows. For clarity, we denote the to-be-extracted variable as *ev*, the parent statement where *ev* is extracted as *pev*, and the homogeneous variable as *hv*. Initially, we created a Variable Dependency Graph (VDG) for each *hv*. A VDG, constructed over the Abstract Syntax Tree (AST) structure, represents the variable dependency relationships, delineating where the variable has been accessed or referenced. An example of VDG is presented in Fig. 4, and its corresponding code snippet is presented in listing 1 (lines 12-20). A VDG is constructed this way:

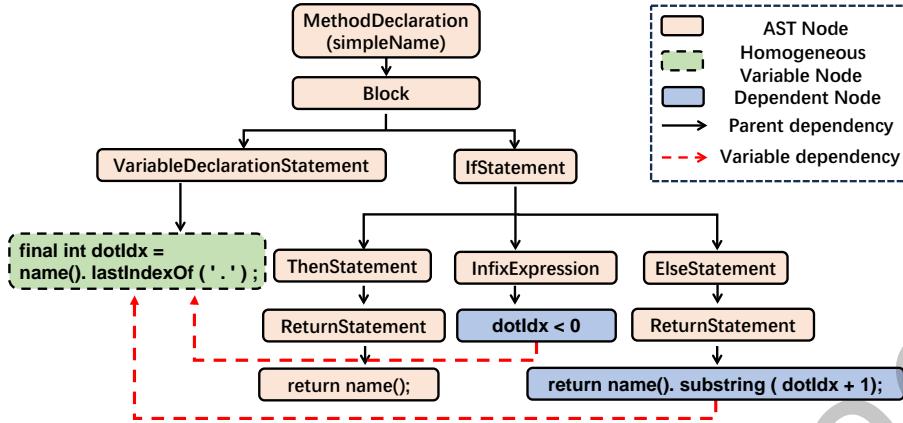


Fig. 4. Variable Dependency Graph

- Initially, we parsed the sibling method where we found  $hv$ , denoted as the *MethodDeclaration* named “*simpleName*”, to generate an abstract syntax tree leveraging JDT [3].
- Subsequently, we identified the AST node declaring  $hv$  and the AST nodes dependent on  $hv$  through static code analysis techniques.
- Finally, we augmented the original AST by adding variable dependency edges in addition to the simple parent-child edges, resulting in a VDG for  $hv$ . As illustrated in Fig. 4, homogeneous variable ( $hv$ ) named “*dotIdx*” is represented by the green box with a dotted border (line 14 in Listing 1). The variable dependent nodes, which are statement-level AST nodes that have a reference (i.e., access) to the  $hv$ , are depicted with blue boxes (lines 15 and 18 in Listing 1). Note that a special case occurs when the variable dependent node appears within a control structure, such as an *if*, *for*, or *while* statement. In these cases, the variable dependent nodes correspond to the conditions of these control structures (i.e., “*dotIdx < 0*” in Fig. 4). These nodes are connected to the  $hv$  via variable dependency edges, represented by a red dotted line.

We then explain how the fine-grained validator works with this example: First, we construct the VDG for  $hv$  (“*dotIdx*”), which is found in the sibling method “*simpleName*”, and traverse the VDG to obtain the dependent nodes via equation 2. Second, we measure the structural similarity (equation 3) and literal similarity (equation 4) of *pev* and each *vdn*. Third, we measure the context similarity by equation 5, and the highest context similarity is selected if there are multiple *vdns*. If the context similarity is greater than a tunable parameter *FGSim*, we consider  $hv$  reliable. Finally, if there are multiple reliable *hvs*, we calculate the method similarity in the same way (equations 3 4 5) and the one with the highest method similarity is selected. Note that the calculation of structural similarity is inspired by Liu et al. [34, 35, 58]. They deemed two AST nodes identical when the AST node type and value were the same. We loosen the logic and deem two AST nodes as identical if the AST node type is the same. The literal similarity is measured by normalized Levenshtein distance [88]. Through calculation, the similarities between two *vdns* and *pev* are 0.24 and 0.67. Finally, we select the latter one (i.e., the highest one) to represent the final context similarity. Due to  $0.67 > FGSim$  (refer Table 4), *VarNamer* will reuse the name of the homogeneous variable (i.e., “*dotIdx*”), which is identical to the name given by the original developers. It is worth noting that the reuse strategy has been submitted as pull requests to the *Eclipse* community, and they have been merged into the mainstream [82, 83].

$$\{\text{Variable Dependent Nodes}\} = \text{Traverse}(VDG(hv)) \quad (2)$$

where  $\text{VDG}(\text{hv})$  denotes the variable dependency graph of  $\text{hv}$ ;  $\text{Traverse}(\ast)$  is a function that traverses the given VDG to obtain all the dependent nodes;

$$\begin{aligned} \forall \text{vdn} \in \{\text{Variable Dependent Nodes}\}, \\ \text{Structural Similarity } (\text{pev}, \text{vdn}) &= \text{dice}(\text{pev}, \text{vdn}) \\ &= 2 * \frac{|\text{nodes}(\text{pev}) \cap \text{nodes}(\text{vdn})|}{|\text{nodes}(\text{pev})| + |\text{nodes}(\text{vdn})|} \rightarrow [0, 1] \end{aligned} \quad (3)$$

where the  $\text{pev}$  and  $\text{vdn}$  are both statement-level AST nodes;  $\text{nodes}(\ast)$  is a function to return the sub-tree associated with the AST node  $\ast$ , more specifically, all descendant nodes of  $\ast$ .  $\text{nodes}(\text{pev}) \cap \text{nodes}(\text{vdn})$  denote the common nodes (i.e., share the same node type).

$$\begin{aligned} \forall \text{vdn} \in \{\text{Variable Dependent Nodes}\}, \\ \text{Literal Similarity } (\text{pev}, \text{vdn}) &= 1 - \frac{\text{Levenshtein}(\text{pev}, \text{vdn})}{\max(\text{length}(\text{pev}), \text{length}(\text{vdn}))} \\ &\rightarrow [0, 1] \end{aligned} \quad (4)$$

where  $\text{Levenshtein}$  is the function to calculate the Levenshtein distance between two texts (i.e., the entire line of the statements without any modifications), and  $\max$  is the function to pick out the larger length of two texts;  $\text{length}$  denotes the character length of the expression.

$$\begin{aligned} \forall \text{vdn} \in \{\text{Variable Dependent Nodes}\}, \\ \text{Context Similarity } (\text{pev}, \text{vdn}) &= \frac{1}{2} * \text{StructuralSimilarity}(\text{pev}, \text{vdn}) \\ &\quad + \frac{1}{2} * \text{LiteralSimilarity}(\text{pev}, \text{vdn}) \\ &\rightarrow [0, 1] \end{aligned} \quad (5)$$

### 4.3 Generation-Based Variable Name Recommendation

In this section, we employ data mining techniques to explore the conventions and patterns associated with variable naming, particularly focusing on the relationship between variable names and their corresponding initializations. Through data mining, we aim to uncover insights into how programmers typically name variables based on how they are initialized. This investigation will provide an understanding of common practices and conventions in variable naming, which facilitates the development of effective naming recommendation tools.

**4.3.1 Data Construction.** We utilized a dataset consisting of 430 open-source projects (GitHub repositories) originally collected by Liu et al. [63] for our mining dataset. These projects were sourced from four prominent communities—Apache, Spring, Hibernate, and Google—and were selected based on having a minimum of 100 commits. This criterion ensures that the projects are well-maintained and likely exhibit a higher quality of variable naming practices. For reference, the list of project names and their corresponding URLs can be accessed on our online repository [33]. We built our dataset for mining as follows:

- To prevent data leakage, we excluded projects utilized in the empirical study (Section 3) and testing (Section 5) from the 430 projects obtained from Liu et al. [63]. This curation resulted in 374 projects, encompassing 326,050 Java files, all of which were utilized for data mining purposes.

- Next, we extracted local variable declarations with initializations from these Java files using JDT [3]. This extraction process yielded a dataset, denoted as *MiningDataSet*, comprising a total of 1,608,843 variable declarations.
- Finally, recognizing that the specific features of various types of initializations may influence variable naming conventions, we partitioned *MiningDataSet* based on the AST node type of the initializations. This segmentation facilitated the subsequent mining procedures.

---

**Algorithm 1** Mine Conventional Naming Rules.

---

**Input:** *MiningDataSet*: The Dataset for Mining.

**Output:** *NamingRules*: The Conventional Naming Rules.

```

1:  $[(NameTokens, IniTokens)] = \text{preprocess}(MiningDataSet)$ 
2:  $\text{Alignments} = \text{AlignAndReplace}([(NameTokens, IniTokens)])$ 
3: for  $Alignment \in \text{Alignments}$  do
4:   if ( $Alignment.\text{contains}(placeholder)$ ) then
5:      $\text{ValidAlignments.add}(Alignment)$ 
6:   end if
7: end for
8: for  $Alignments \in \{\text{ValidAlignments}\}$  do
9:    $FPTree = \text{CreateTree}(Alignments, MinSupport)$ 
10:   $\text{CandidateRules} = \text{GenerateAssociationRules}(FPTree, MinConfidence)$ 
11:   $\text{NamingRules} = \text{Validate}(\text{CandidateRules})$ 
12: end for
```

---

4.3.2 *Mining Conventional Naming Rules*. In this section, we investigated the conventional rules developers tend to follow when naming variables based on their initializations by employing association analysis techniques. The mining process is outlined in Algorithm 1. Here's a step-by-step breakdown:

- **Preprocessing:** We preprocessed the variable names and their initializations using the following steps:
  - We utilized javalang [7] to extract only identifier tokens from variable names and their initializations.
  - We used spacy [46] to split the identifier tokens into sub-tokens, resulting in a 2-tuple [*NameTokens*, *IniTokens*] (line 1).
- **Alignment:** For each 2-tuple, we aligned *NameTokens* and *IniTokens* to identify the sub-tokens that appear in both sets. Subsequently, we replaced the aligned sub-tokens in initializations with a placeholder, excluding variable names consisting of only one letter (line 2). For instance, if we had an initialization “*checkConfig.getMessages()*” and its corresponding variable name “*messages*”, the alignment resulted in *alignment*: [“*check*”, “*config*”, “*get*”, *placeholder*].
- **Validation:** We reserved cases where the alignments and replacements succeed, resulting in a collection *ValidAlignments* (lines 3-7).
- **Mining with FP-growth:** We employed the FP-growth algorithm [44] (lines 8-12), known for its efficiency and reliability in association rules mining [50]. The algorithm mined frequently co-appearing items (*FCI*) in *ValidAlignments*, yielding *CandidateRules*. It's worth noting that initializations with different AST node types were analyzed separately.
- **Manual Validation:** We manually validated the *CandidateRules* to ensure their reasonability, resulting in the final *NamingRules*. The parameters *MinSupport* and *MinConfidence* were configurable and determined the minimum frequency of items in the *FPTree* and the conditional probability of an *FCI*. These parameters are discussed in detail in Section 5.5.

Some examples of the finally obtained naming rules are as follows (only identifier tokens are preserved):

$$<\text{placeholder}> = \text{fetch} + <\text{placeholder}>; \quad (6)$$

$$<\text{placeholder}> = \text{generate} + <\text{placeholder}>; \quad (7)$$

$$<\text{placeholder}> = <\text{placeholder}>s + \text{next}; \quad (8)$$

$$<\text{placeholder}> = <\text{placeholder}>es + \text{next}; \quad (9)$$

where  $<\text{placeholder}>$  denotes the tokens of variable names, and “+” concatenates the two frequently co-occurring items. The application conditions of the first two rules are two-fold: First, the initialization should be a method call expression; Second, the method name should start with “fetch” or “generate”. If these two conditions are satisfied, the variable name is likely the nouns that follow the starting verbs. For example, *VarNamer* will recommend “urls” for the initialization “*generateUrls(String names)*”, and “*executionStatus*” for the initialization “*fetchExecutionStatus()*”. There are three application conditions of the last two rules: First, the initialization should be a method call expression; Second, the method name should be “next”; Last, the receiver should be a plural name. If these three conditions are satisfied, the variable name is likely the singular form of the receiver. For example, *VarNamer* will recommend “*feature*” for the initialization “*features.next()*”, and “*alias*” for the initialization “*aliases.next()*”.

It is worth noting that the mined rules have been submitted as pull requests [81, 84] to the *Eclipse* community, and the pull requests have been merged.

#### 4.4 Selection and Recommendation

The names recommended by the name reuse component (*ReusedNames*) and name generation component (*GeneratedNames*) are collected, resulting in a name list, *NameList*. The name selection component works in this way:

- We first validate names in *NameList* to identify invalid entries. We consider names invalid if they fall into either of the following categories:
  - Java keywords, e.g., class, for, if, static, int.
  - Names already used in the enclosing block, e.g., local variable names or parameter names of methods.
 Any names falling into these categories are removed from *NameList*.
- Following validation, if *NameList* contains only one name, either *ReusedName* or *GeneratedName* is recommended. In the case where *NameList* contains two names, i.e., a *ReusedName* and a *GeneratedName*, the *ReusedName* is prioritized due to the higher precision associated with the name reuse component (refer to Section 5.7).

Using the code snippets presented in Listing 1, let’s walk through how *VarNamer* recommends a name for the extract local variable refactoring:

- *Homogeneous Variable Retrieval*: The reuse component retrieves a homogeneous variable named “*dotIdx*”. Through the filter and validator, this homogeneous variable is considered reliable for reuse.
- *Name Generation*: Simultaneously, the generation component generates a name based on the initialization. In this case, it generates the name “*lastIndexOf*”.
- *Name Validation*: Both “*dotIdx*” and “*lastIndexOf*” are validated to ensure they are valid names. This step verifies whether the names are not Java keywords and are not already used within the enclosing block.
- *Final Recommendation*: After validation, “*dotIdx*” is selected from *ReusedNames* and *GeneratedNames* as the final recommendation. This decision is based on the fact that *VarNamer* prioritizes reused names over generated ones.

By following these steps, *VarNamer* ensures that the recommended name for the extracted local variable is both valid and based on established conventions or contextual information.

## 5 EVALUATION

### 5.1 Research Questions

- **RQ3:** How does *VarNamer* perform in recommending names for extracted variables compared with baselines?
- **RQ4:** How do the major components contribute to the performance of *VarNamer*?
- **RQ5:** How does *VarNamer* perform regarding time efficiency?
- **RQ6:** How does *VarNamer* work on programming languages other than Java?
- **RQ7:** To what extent can *VarNamer* aid developers in conducting extract local variable refactoring?

RQ3 focuses on evaluating the effectiveness of *VarNamer* compared to selected baselines in recommending names for the extract local variable refactoring tasks. Additionally, we aim to uncover the underlying factors contributing to *VarNamer*'s superior performance over the baselines. By addressing RQ3, we gain insights into how *VarNamer* performs in real-world scenarios and the specific areas where it excels compared to baseline approaches. Understanding the advanced improvements of *VarNamer* can inform future enhancements and optimizations to enhance its efficacy further.

RQ4 aims to determine the extent to which the major components of *VarNamer*, namely name reuse, name generation, and name selection, contribute to its overall performance. By dissecting the roles and effectiveness of these components, we gain insights into the inner workings of *VarNamer* and its capacity to recommend satisfactory variable names. Understanding the individual contributions of these components is crucial for optimizing *VarNamer* and refining its functionality to better serve developers' needs in code refactoring tasks.

RQ5 focuses on evaluating the time efficiency of *VarNamer* in comparison to baseline approaches, aiming to determine whether *VarNamer* can meet the in-time needs of integrated development environments (IDEs) and developers. In addressing RQ5, we employ specific metrics, i.e., the time taken to recommend variable names, to gauge the time efficiency of *VarNamer* and the baselines. By comparing the performance of *VarNamer* against baseline approaches, we aim to elucidate its ability to deliver timely and efficient variable name recommendations, thereby supporting seamless and productive software development workflows.

RQ6 delves into the applicability of *VarNamer* beyond the Java programming language and aims to determine whether its performance remains consistent across different programming languages. In this research question, we evaluate *VarNamer* in languages beyond Java, e.g., C++. By investigating RQ6, we may reveal its potential as a language-agnostic tool for enhancing developer productivity.

Having examined the performance, efficacy, and extensibility of *VarNamer* in recommending variable names through preceding research questions, we now turn our focus to RQ7. While accurate recommendations are essential, their true value lies in their practical utility for developers. We consider two metrics to gauge the effectiveness of *VarNamer* in assisting developers. These metrics encompass factors such as time savings during refactoring tasks, and the reduction in manual effort required for variable naming. By investigating RQ7, we may reveal the real-world impact of *VarNamer* on developers' productivity.

### 5.2 Dataset

To evaluate the performance of *VarNamer* and the baselines in recommending names for the extract local variable refactorings, we constructed a real-world refactoring dataset, called *TestingDataSet*, as specified in Section 3.2.1. The dataset contains 27,158 real-world extract local variable refactorings that were mined automatically by RefactoringMiner [78] from the top 1,000 Java open-source projects (sorted by stars) in GitHub. This selection of projects was made to ensure a diverse and representative sample of real-world refactorings. Notably, to guarantee the high quality of the resulting dataset, we designed a series of rules to filter out the false positives of RefactoringMiner. We would investigate research questions RQ3-5 with this dataset.

To investigate RQ6, i.e., how the proposed approach works on programming languages other than Java, we should build another dataset by collecting real-world extract local variable refactorings from source applications in other programming languages, e.g., C++. However, to the best of our knowledge, there are no automatic tools that could discover the extract local variable refactorings in C++ applications. To this end, we manually constructed a dataset for C++ (noted as *C++Dataset*), which also serves as the dataset for investigating RQ7. The process of the data construction is as follows:

- First, we selected five well-known open-source projects from Google and Apache (i.e., “arrow” [17] and “mesos” [18] from Apache, and “angle” [39], “dawn” [40], and “skia” [41] from Google) on GitHub. These five projects are all active and well-maintained (with over 15,000 commits). In addition, they are across different domains including graphics engine, webGPU implementation, cluster manager, and development platform.
- Second, we tracked the commit history of the selected projects and analyzed the hunks of each commit manually to collect the extract local variable refactorings.
- Note that the manual analysis is time-consuming and labor-intensive. To this end, for each project, we collected 10 extract local variable refactorings in the order of their appearance, resulting in 50 real-world refactorings in total. Note that there may be multiple extract local variable refactorings targeting the same expression in a single commit. To improve the quality of the dataset, we only kept two instances of them.

As a conclusion, we have constructed two datasets for the evaluation, i.e., *TestingDataSet* containing 27,158 extract local variable refactoring in Java applications, and *C++Dataset* containing 50 extract local variable refactoring in C++ applications.

### 5.3 Baseline Approaches

We included the IDEs and large language model (LLM) approaches as our baselines. For IDE baselines, we selected *Eclipse* [1] and *IDEA* [6] due to their widespread usage in the software development community and their comprehensive features for code editing and refactoring. We implemented the extract local variable refactoring function of *Eclipse* and *IDEA* by leveraging their plugin development framework where the internal name recommendation interface is available. Since *IDEA* provides a list of names for developers, we selected the first name from the list as the final choice to ensure consistency and fairness in the comparison process. For LLM approaches, we selected *InCoder* [36], which represents one of the state-of-the-art approaches in code completion, particularly focusing on cloze-style inference techniques. Note that although *InCoder* was originally evaluated in Python programs, it supports variable name prediction tasks in multiple programming languages, and its performance on the cloze task in Java language is even better than that in Python language as reported by Fried et al. [36]. Consequently, it is suitable to take *InCoder* as our baseline on variable name recommendation in Java language. We utilized the Python implementation of the *InCoder-6.7B* model obtained on Hugging Face [5], a widely recognized platform for accessing and sharing pre-trained models and libraries for natural language processing tasks. To evaluate the performance of *InCoder*, we prepared the data fed into *InCoder* as follows:

- First, for each refactoring in *TestingDataSet*, we obtained the Java file where the extract local variable refactoring happened (after the refactoring).
- Second, we further located the methods enclosing the refactoring and replaced the names of the newly introduced variables and their references with “*<infill>*” tokens, as required by *InCoder*. This preprocessing step ensures that the code snippets are formatted correctly for input into *InCoder*, facilitating accurate variable name prediction.
- Third, we input the preprocessed code snippets, containing cloze-style placeholders, into *InCoder*. We extracted the generated name for the first “*<infill>*” token, which corresponds to the variable declaration, from the output. For instance, in the code snippet “*final int <infill> = name().lastIndexOf();*”, the infilled

Table 4. Settings of *VarNamer*

Parameter	Value
ProjectNum	80
FGSim	0.3
MinConfidence	0.8
IniLength	30
MinSupport	50

name generated by *Incoder* for the placeholder was considered as the final name for comparison. This also maintains consistency in the evaluation methodology.

#### 5.4 Performance Metrics

To measure the performance of the evaluated approaches, we adopted a series of metrics:

- *#Total Cases*: the number of cases involved in the evaluation, i.e., the frequency of invocation of the evaluated approaches.
- *#Recommendation*: the number of cases where the evaluated approaches make a recommendation for the developers. It is worth noting that when *VarNamer* fails to generate a reasonable name (i.e., both name reuse and name generation components failed to suggest any variable name), it opts to make no recommendation rather than offer a low-quality one. That is to say, *VarNamer* will not make recommendations to all the data (#total cases) as the other three baselines. Therefore, for *VarNamer*, the number of recommendations (#recommendation) may not always match the total number of cases (#total cases). This discrepancy should be considered when interpreting the results of the evaluation.
- *#Exact Match*: the number of cases where the recommended variable names are identical to the ground truth (i.e., the expected names). Notably, achieving an exact match is crucial because in such cases, developers can readily accept the recommendations without the need for additional edits. This not only streamlines the coding process but also ensures consistency and conciseness in the code base [31]. *#Exact match* should ideally be equal to or less than *#recommendation* as an evaluated approach can only achieve an exact match if it makes a recommendation for the given case.
- *EMPrecision*: the number of exact matches divided by the number of recommendations, i.e.,

$$EM_{Precision} = \frac{\#Exact\ Match}{\#Recommendation} \quad (10)$$

*EMPrecision* presents how often the suggested variable names are correct. The more often the suggested names are incorrect (i.e., with lower *EMPrecision*), the more likely developers may abandon the suggestion approach/tool.

- *EMCoverage*: the number of exact matches divided by the number of cases involved in the evaluation, i.e.,

$$EM_{Coverage} = \frac{\#Exact\ Match}{\#Total\ Cases} \quad (11)$$

*EMCoverage* presents the ratio of exact match cases generated by each suggestion approach out of the total number of cases.

#### 5.5 Setup

For a fair comparison, we independently evaluated each of the involved approaches, including the proposed approach and the baselines, on the same server. The setting of the server is as follows:

Table 5. Improving the State of the Art

Metrics	Eclipse	IDEA	Incoder	VarNamer
#Recommendation	27,158	27,158	27,158	21,766
#Exact Match	7,380	8,022	9,006	9,036
$EM_{Precision}$	27.2%	29.5%	33.2%	41.5%
$EM_{Coverage}$	27.2%	29.5%	33.2%	33.3%

Table 6. Performance Comparison in *CommonDataSet*

Metrics	Eclipse	IDEA	Incoder	VarNamer
#Recommendation	21,766	21,766	21,766	21,766
#Exact Match	7,052	7,509	7,640	9,036
$EM_{Precision}$	32.4%	34.5%	35.1%	41.5%
$EM_{Coverage}$	32.4%	34.5%	35.1%	41.5%

- Operation system: Ubuntu 18.04.1;
- CPU: 32 \* Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz;
- GPU: 4\*TITAN RTX (used by the baseline approach *Incoder* only);
- RAM: 128 GB.

*VarNamer* has several parameters that need to be tuned. We first quantified the valid value range for parameters such as *ProjectNum* ([0, 374]), *InitLength*([0, 66]), and *FGSim*([0.0, 1.0]). The maximum number of projects is all the projects included in *MiningDataSet*, i.e., 374. For *InitLength*, we determined the upper bound by considering the outliers' upper bound of character length in the *EmpiricalDataSet*, calculated as  $Q3+1.5*IQR=66$ , to ensure an appropriate parameter range. The *FGSim* is calculated by equation 5 which outputs a decimal between 0 and 1. The step sizes for *ProjectNum* and *InitLength* were set to 10, while the step size for *FGSim* was set to 0.1. These step sizes were chosen to facilitate a systematic exploration of parameter values. Subsequently, we employed the grid searching algorithm with the specified value ranges and step sizes to identify the parameter values that optimize the  $EM_{Precision}$  and  $EM_{Coverage}$  of *VarNamer*. *MinSupport* controls the number of considered frequency items. With the increase of *MinSupport*, considered frequency items decrease, leading to less useful co-appearing relationships and less time cost. *MinConfidence* decides the reliability of the mined relationships. As a result, we empirically tuned the two parameters to make a balance between the time cost and the mined results. The finally adopted parameters are presented in Table 4.

## 5.6 RQ3: Improving the State of the Art

To answer RQ3, we evaluated the selected approaches on *TestingDataSet* independently. The evaluation results are presented in Table 5. The second to fourth columns in Table 5 present the performance of the selected baselines on the given testing data (i.e., *TestingDataSet*). The last column presents the performance of the proposed approach, i.e., *VarNamer*. The second row denotes the number of cases where the evaluated approaches made suggestions whereas the third row presents the number of cases where the variable names suggested by the evaluated approaches are identical to the ground truth. The fourth and the fifth rows present the  $EM_{Precision}$  and  $EM_{Coverage}$ , respectively.

From Table 5 we make the following observations:

- *VarNamer* may refuse to recommend in some cases while the baselines will recommend in all the cases. Although *VarNamer* makes the fewest number of recommendations, i.e., 21,766 vs. 27,158, the number of its exact-match names is the highest, i.e., 9,036 vs. 7,380 for *Eclipse*, 8,022 for *IDEA*, and 9,006 for *Incoder*.
- *VarNamer* outperforms all the baseline approaches by a significant margin. Compared with one of the state-of-the-art methods in recommending variable names, i.e., *Incoder*, *VarNamer* improves the  $EM_{Precision}$  by  $25.0\% = (41.5\% - 33.2\%) / 33.2\%$  without sacrificing  $EM_{Coverage}$ .
- Compared with the current implementations of popular IDEs such as *Eclipse* and *IDEA* for recommending variable names, *VarNamer* significantly improves the  $EM_{Precision}$  by  $52.6\% = (41.5\% - 27.2\%) / 27.2\%$  (compared to *Eclipse*) and  $40.7\% = (41.5\% - 29.5\%) / 29.5\%$  (compared to *IDEA*).

Since *VarNamer* works on a subset of 21,766 instances, we also report how the other baselines perform on this same subset, i.e., *CommonDataSet* in Table 6. The table structure of Table 6 is the same as Table 5. From Table 6 we make the following observations:

- When evaluated on the same sub-dataset, i.e., *CommonDataSet*,  $EM_{Precision}$  and  $EM_{Coverage}$  are identical for all the four approaches since the number of recommendations is equal to the number of total cases.
- *VarNamer* can still achieve the best  $EM_{Precision}$  and  $EM_{Coverage}$ , i.e., 41.5%, compared to the other three baselines, outperforming *Incoder* in  $EM_{Precision}$  and  $EM_{Coverage}$  by 6.4 percentage points.
- The  $EM_{Coverage}$  of *VarNamer* improves from 33.3% to 41.5% as the number of recommendations decrease.
- The  $EM_{Precision}$  and  $EM_{Coverage}$  of *Eclipse* and *IDEA* improve by 5.2 percentage points and 5.0 percentage points, respectively while *Incoder* only improves by 1.9 percentage points.

To investigate the reason why *VarNamer* outperforms the baseline approaches, we conducted a data analysis, and the results suggest that there are 2,260 and 2,267 cases where *VarNamer* succeeded in recommending a satisfying name while *Eclipse* and *IDEA* failed, respectively. 81.3% ( $=1,838/2,260$ ) of the names recommended by *Eclipse* is deemed sub-optimal due to the absence of the reuse component present in *VarNamer*. The remaining 18.7% ( $=422/2,260$ ) of the recommended names are sub-optimal because of the missing generation rules. For *IDEA*, 77.5% ( $=1,757/2,267$ ) of the recommended names are sub-optimal due to the absence of the reuse component, while 22.5% ( $=510/2,267$ ) are sub-optimal due to missing generation rules. We illustrate the advancement of the reuse component with listing 1 in Section 2. In this example, *VarNamer* retrieved a reliable homogeneous variable from the sibling method in the same Java file and reused its name. Consequently, *VarNamer* recommended “*dotIdx*”, which is identical to the name given by the original developers. However, *Eclipse* recommended “*lastIndexOf*” and *IDEA* recommended “*x*”, which are both sub-optimal. The advancements of the generation component in *VarNamer* encompass two key aspects: First, to recommend names for method calls, both *Eclipse* and *IDEA* take the heuristic rule which removes the popularly starting verbs of the method names and only keeps the trailing content. To ensure time efficiency, *Eclipse*, and *IDEA* employ a heuristic approach to retain a predefined set of popularly used verbs, such as “*get*”, “*is*”, “*to*” for *Eclipse* and “*get*”, “*is*”, “*to*”, “*find*”, “*create*”, “*as*” for *IDEA*, respectively. However, through the mining process detailed in Section 4.3.2, we discovered a significant number of popular verbs commonly used in programming, such as “*read*”, “*build*”, “*add*”, “*parse*”. This expanded understanding of common programming verbs enhances *VarNamer*’s ability to recommend variable names that are preferred by developers. For example, for initialization “*buildId(serviceInstance)*” [32], the involved method declaration is presented in Listing 2. As we can see from this code snippet, the functionality of the method “*buildId*” is to generate a unique ID for a service object and return it. As a result, developers prefer the variable name ‘*id*’ in this context. For this case, *VarNamer* removed the verb “*build*” and recommended “*id*” as the variable name, aligning better with the developer preferences. However, *Eclipse* recommended “*buildId*” and *IDEA* recommended “*s*”, both of which are sub-optimal. Additionally, *VarNamer* excels in cases where initializations involve accessing elements from a collection. For example, for initialization “*features.next()*” [37], the involved method declaration

```

1 public void unregister(ServiceInstance serviceInstance) throws RuntimeException {
2 -   client.agentServiceDeregister(buildId(serviceInstance));
3 +   String id = buildId(serviceInstance);
4 ...
5 +   client.agentServiceDeregister(id);
6 }
7 private String buildId(ServiceInstance serviceInstance) {
8     // let's simply use url's hashCode to generate unique service id for now
9     return Integer.toHexString(serviceInstance.hashCode());
10}

```

Listing 2. Extract Local Variable Refactoring from ConsulServiceDiscovery.java

```

1 public void writeInternal(AbstractItemsResponse itemsResponse, HttpOutputMessage
    httpOutputMessage) throws IOException {
2     try (FeatureIterator features = itemsResponse.getItems().features()) {
3         while (features.hasNext()) {
4             builder.evaluate(writer, new TemplateBuilderContext(features.next()));
5             // lookup the builder, might be specific to the parent collection
6             Feature feature = features.next();
7             ...
8             builder.evaluate(writer, new TemplateBuilderContext(feature));
9         }
10    }
11}

```

Listing 3. Extract Local Variable Refactoring from TemplatedItemsConverter.java

is presented in Listing 3. As we can see from this code snippet, “*next*” is a method of “*FeatureIterator*”, and it returns an element from a collection. Consequently, “*feature*”, which is the singular form of “*features*”, is preferred by developers. With an exploration of real-world refactoring data, *VarNamer* correctly identified “*feature*” as the preferred variable name. However, *Eclipse* and *IDEA* both recommended “*next*” as the variable name, which is sub-optimal.

To investigate why *VarNamer* outperforms *Incoder*, we randomly sampled 356 methods from 4,767 cases where *VarNamer* succeeded in recommending a satisfying name while *Incoder* failed, with a confidence level of 95% and a margin of error of 5%. Two authors independently inspected the sampled cases and tried to find out any possible reasons that could explain why sometimes *Incoder* failed and *VarNamer* succeeded in recommending a correct name. The Cohen’s kappa coefficient of agreement between the two authors is 0.82. Any discrepancies were resolved through discussion until a consensus was reached. Compared to *VarNamer*, *Incoder* may not thoroughly leverage the *initialization* context, which is essential in recommending a name for it. For example, on 299 out of the 356 samples, the extracted expressions are method invocations, and thus method invocations were used as initialization for the newly introduced variables. In such cases, our approach successfully suggested the expected variable name by extracting tokens from method names in the extracted method invocation. However, *Incoder* failed in a majority of cases (161 out of 299). We also noticed that such cases (161) accounted for nearly half of the analyzed samples (356 in total), suggesting that they pose significant challenges for *Incoder*. For example, for initialization “*checkConfig.getMessages()*” [25], the involved method declaration is presented in Listing 4. As we can see from the code snippet, The functionality of the method “*getMessages*” is to return a *HashMap* Object containing custom messages. Consequently, “*messages*” here is preferred by developers. However, *Incoder* recommended “*project*” even though “*messages*” can be simply extracted from the method name. This challenge

```

1 public void testTypeName() throws Exception {
2     ...
3     final Map<String, String> messages = checkConfig.getMessages();
4     "71:12: " + getCheckMessage(checkConfig.getMessages(), msgKey, "Annotation$", format),
5     "71:12: " + getCheckMessage(messages, msgKey, "Annotation$", format),
6     ...
7 }
8 /** The map containing custom messages. */
9 private final Map<String, String> messages = new HashMap<>();
10 ...
11 /* @return unmodifiable map containing custom messages */
12 @Override
13 public Map<String, String> getMessages() {
14     return new HashMap<>(messages);
15 }

```

Listing 4. Extract Local Variable Refactoring from TypeNameTest.java

Table 7. Contributions of VarNamer’s Major Components

	$EM_{Precision}$	$EM_{Coverage}$
VarNamer	41.5%	<b>33.3%</b>
w/o Name Generation	<b>80.3%</b>	10.7%
w/o Name Generation [Additional Rules]	40.8%	30.2%
w/o Name Reuse	34.9%	27.2%
w/o Name Selection	35.3%	28.8%

may be solved by fine-tuning *Incoder* with our data. However, since *Incoder* is trained on high-performance GPU devices (such as dual GPUs, which are not typically available to most developers), its performance in real-world scenarios would likely be even worse than what is reported. Consequently, we did not fine-tune *Incoder* in this paper. It is also worth noting that *Incoder* may be inflated due to data leakage. To figure out the situation of data leakage, we manually checked the involved projects, and the results suggest that 77.3% (=576/745) of the projects used in this paper appeared in the training data of *Incoder*. Despite the data leakage, VarNamer still outperforms *Incoder*, demonstrating its effectiveness.

VarNamer’s failure to recommend satisfying names can be attributed to the imperfect  $EM_{Precision}$  in both the reuse and generation components (80.3% for the reuse component and 34.9% for the generation component). Furthermore, VarNamer cannot incorporate developers’ naming preferences, such as the length (prefer shorter or longer names) and format (prefer full names or abbreviations) of variable names. Integrating these preferences is essential for ensuring that recommended names align with developers’ expectations and coding conventions.

*Answer to RQ3:* VarNamer significantly improves the  $EM_{Precision}$  by 52.6% (compared to Eclipse) and 40.7% (compared to IDEA). Compared with *Incoder*, VarNamer improves the  $EM_{Precision}$  by 25% without sacrificing  $EM_{Coverage}$ .

### 5.7 RQ4: Contributions of Major Components of VarNamer

To address this research question, we removed each component successively, resulting in three variants of *VarNamer*. We then evaluated each variant's performance on the *TestingDataSet* and compared it to the baseline performance of *VarNamer* with all components enabled. To investigate the influence of the name selection component, we removed the name validation and adopted a random selection strategy to select the final name for recommendation. To figure out the real contributions of the additional rules proposed in this paper, we also excluded the additional rules mined from a large code corpus to avoid potential overlap with the existing rules adopted by *Eclipse* and *IDEA*.

The evaluation results presented in Table 7 reveal the following observations:

- All three major components make significant contributions to the performance of *VarNamer*, as evidenced by the significant decrease in performance when any of them is removed.
- The name reuse component contributes more to the  $EM_{Precision}$ , with a decrease of  $15.9\% = (41.5\%-34.9\%)/41.5\%$  in  $EM_{Precision}$  when it is removed. The  $EM_{Precision}$  of *VarNamer* remains high at 80.3% even when the name generation component is removed, providing additional support for the significant contribution of the name reuse component.
- The name generation component contributes more to the  $EM_{Coverage}$ , with a decrease of  $67.9\% = (33.3\%-10.7\%)/33.3\%$  when it is removed. The additional rules in this component also play a crucial role, as demonstrated by a decrease of  $9.3\% = (33.3\%-30.2\%)/33.3\%$  in  $EM_{Coverage}$  when these rules are removed.
- The name selection component is of great importance to *VarNamer*: when it is removed, the  $EM_{Precision}$  decreases by  $14.9\% = (41.5\%-35.3\%)/41.5\%$  and the  $EM_{Coverage}$  decreases by  $13.5\% = (33.3\%-28.8\%)/33.3\%$ .

It is worth noting that we did not incorporate data type in our approach although data type is one of the most critical contexts where name tokens of variables can be found. We investigated the impact of data type on the *VarNamer*'s performance and the results are presented in Table 8. Based on the current implementation of *VarNamer*, we further developed two variants, i.e., "w/ Data Type" and "w/ Data Type + Selection Strategy". These two variants indicate incorporating data types without any selection strategy and with a selection strategy, respectively. The selection strategy involves making recommendations only for cases where the types are customized by developers, as our observations suggest that developers are highly likely to use user-defined types directly as variable names.

From Table 8, we make the following observations:

- Incorporating data type into *VarNamer* without any selection strategy improves  $EM_{Coverage}$  by 1.6 percentage points, while  $EM_{Precision}$  significantly drops by 5.9 percentage points.
- With the selection strategy,  $EM_{Coverage}$  still improves by 1.5 percentage points, while the  $EM_{Precision}$  drops by 1.2 percentage points. A trade-off exists between  $EM_{Coverage}$  and  $EM_{Precision}$  when incorporating data type into *VarNamer*.
- Ideally, adding more effective selection strategies could further improve the  $EM_{Coverage}$  without sacrificing too much  $EM_{Precision}$ . However, in this paper, we choose to not incorporate data type because we prioritize  $EM_{Precision}$ .

The possible reason that adding data types reduces  $EM_{Precision}$  is due to their high overlap with other types of contexts, even though the chance of an exact match between data types and the expected variable names is 13.8% as shown in Table 2. Our investigation found that 80.9% of sub-tokens found in *data types* also appear in *initializations*. Additionally, the overlapping with *homogeneous variables* and *assignments* are 16.5% and 0.9%. Overall, only 6.2% sub-tokens can be found exclusively in *data types*.

Table 8. Impact of data type on the VarNamer's Performance

Setting	$EM_{Precision}$	$EM_{Coverage}$
VarNamer	41.5%	33.3%
w/ Data Type	35.6%	34.9%
w/ Data Type + Selection Strategy	40.3%	34.8%

*Answer to RQ4:* All three major components make significant contributions to the performance of *VarNamer*.  $EM_{Precision}$  benefits more from the name reuse component whereas  $EM_{Coverage}$  benefits more from the name generation component. Incorporating data types improves the  $EM_{Coverage}$  with a sacrifice of  $EM_{Precision}$ .

## 5.8 RQ5: Time Efficiency

In this research question, we investigated the time cost of *VarNamer* and the three baselines on *TestingDataSet*, and the comparison results are presented in Table 9. The second column denotes the total running time cost. The third column denotes the average running time cost for a single refactoring, and the last column denotes the median value of the running time cost for a single refactoring. It's important to note our methodology: we ran all the approaches (except *Incoder*) five times repeatedly and took an average of the total time, the average time per refactoring, and the median time per refactoring. Since the name recommendation of *Incoder* is time-consuming, and there is nothing comparable between the time cost of *Incoder* and that of the other three approaches, we only executed *Incoder* once and recorded its time cost for one run. From Table 9, we make the following observations:

- Considering the average time cost, *Incoder* [36] takes the most time (48.4s) to generate a name for an extracted local variable, while *VarNamer*, *Eclipse* [1], and *IDEA* [6] take much shorter time (5.7ms, 0.5ms, and 3.9ms). Concerning the median value, the time cost is 27.0s vs. 5.0ms, 0.2ms, and 3.0ms. This is reasonable because *Incoder* depends on deep neural networks that possess extremely complex structures and billions of parameters (6.7 billion), while the other three approaches utilize efficient heuristic rules based on lexical and syntactic patterns.
- It takes *VarNamer* an average of 5.7ms to recommend a name, which is comparable to that of *IDEA* (3.9ms). This indicates that *VarNamer* is efficient enough to be incorporated into popular IDEs.

In conclusion, we underscore the importance of considering both efficiency and accuracy when evaluating name recommendation approaches. While deep learning-based methods like *Incoder* may offer high accuracy, their efficiency may be compromised. Conversely, the proposed approach *VarNamer* strikes a balance between efficiency and accuracy, making it a suitable candidate for integration into mainstream IDEs.

*Answer to RQ5:* *VarNamer* is efficient. It can make a recommendation in 5ms, comparable to *IDEA*.

## 5.9 RQ6: Application to C++

The experiment results presented in Section 5.6 have indicated that *VarNamer* performs well in recommending variable names in Java programming languages. However, it remains unknown whether *VarNamer* can also

Table 9. Efficiency of Evaluated Approaches

Approaches	Execution Time	Execution Time per Refactoring (Average)	Execution Time per Refactoring (Median)
Eclipse	11.5s	0.5ms	0.2ms
IDEA	84.7s	3.9ms	3.0ms
Incoder	292.6h	48.4s	27.0s
VarNamer	123.5s	5.7ms	5.0ms

*ms*: milliseconds; *s*: seconds; *h*: hours.

be applied to recommend variable names in other programming languages besides Java. To investigate the extensibility of *VarNamer*, we evaluated its performance in C++ programming languages in this research question.

Theoretically, *VarNamer* is language-agnostic because the factors (i.e., homogeneous variables, initializations of variables) are common to most programming languages. However, to the best of our knowledge, there is not a universal AST parser for all programming languages. Due to the absence of a universal AST parser, we had to implement a C++ version of *VarNamer* using an AST parser specifically designed for C++. Our approach was developed based on the *Eclipse CDT* [2], which provides a fully functional C and C++ Integrated Development Environment on the *Eclipse* platform. By incorporating the core logic of our approach into this framework, we obtained the C++ version of our approach and called *VarNamer-C++* for convenience. We evaluate *VarNamer-C++* on C++Dataset, and the process of the dataset construction is presented in Section 5.2.

We leveraged the same metrics, i.e.,  $EM_{Precision}$  and  $EM_{Coverage}$  (refer Section 5.4) to evaluate how *VarNamer-C++* performed in recommending variable names on C++Dataset. The evaluation results suggest that the  $EM_{Coverage}$  and  $EM_{Coverage}$  of *VarNamer-C++* in recommending variable names are both 44.0% (=22/50), which is comparable to the performance (41.5%) in Java programming language. However, the  $EM_{Precision}$  and  $EM_{Coverage}$  of *Eclipse CDT* is only 12.5% (=3/24) and 6.0% (=3/50).

To investigate the reason why *VarNamer-C++* outperforms *Eclipse CDT*, we analyzed the names recommended by them in 50 refactorings. The analysis results suggest that there are two major reasons. One reason is that *Eclipse CDT* declined to recommend a name in most cases (26 out of 50). We found that the recommendation logic of *Eclipse CDT* for the extract local variable refactoring is quite simple and only covers a few specific cases, leading to a low ratio of recommendations. Another reason also exists in the current implementations of IDEs such as *Eclipse* and *IDEA*. As introduced in Section 1, these IDEs ignore many useful contexts, e.g., homogeneous variables, and lack in-depth analysis of real-world refactoring data. Through analysis of real-world refactoring data, *VarNamer-C++* is better equipped to cover more cases and provide recommendations that are more satisfying to developers.

Consequently, we conclude that our approach can be extensively applied to other programming languages besides Java, e.g., C++, and its performance remains stable as indicated by the experiment results.

By reusing the parameter values from the Java version of our approach, as shown in Table 4, we still achieved comparable performance in recommending names for the extract local variable refactorings in C++. However, in our investigation of the differences between Java and C++ programming languages, we observed several distinctions between Java and C++ that may have a slight impact on name recommendation. For instance, in addition to the Java-style method invocation using ‘‘, C++ offers two alternative methods: the pointer operator ‘->’ and the scope operator ‘::’. Notably, the latter two operators consist of one more letter than a single ‘.’. Moreover,

the concept of “*namespace*” is crucial in C++ programming, leading to a prevalent use of scope operators like “*gl::FromGLenumgl::ShadingRate(rate)*”. Developers in C++ often prefix object references with their corresponding namespace. These differences result in longer initialization (in character length) in C++ compared to Java.

*Answer to RQ6:* The performance of *VarNamer* on C++ source code is comparable to that on Java code.

### 5.10 RQ7: User Study

To investigate to what extent *VarNamer-C++* can enhance the efficiency of extract local variable refactoring in the wild, we conducted a user study with real-world extract local variable refactorings (i.e., *C++Dataset*) and *VarNamer-C++*. Characteristics of the refactoring dataset and *VarNamer-C++* are presented in Section 5.2 and Section 5.9, respectively.

We invited six developers who had development experience in C++ projects to conduct the user study. The participants were divided into two groups with three developers in each group. Both groups were asked to conduct extract local variable refactorings on *C++Dataset*, and the difference is that the first group finished it with the help of the standard CDT plugin (latest version by the time we submitted this paper, i.e., *cdt-11.5.0*) and the second group finished it with the help of *VarNamer-C++*. The procedure for developers conducting a single refactoring is as follows:

- For each refactoring instance in *C++Dataset*, participants were asked to first call the name recommendation dialog through shortcut keys “*Alt + Shift + L*” in *Eclipse*.
- Then the participants were asked to judge whether the recommended name was proper in the given context (i.e., the enclosing method declarations).
- If so, they clicked “OK” and finished a single refactoring. Otherwise, they were asked to edit the recommended name until they were satisfied with it. In addition, if the dialog did not contain a name, i.e., no recommendation is available, they were asked to coin a name from scratch for the newly introduced variable. After giving a name to the newly introduced variable, the participants clicked “OK” and finished a single refactoring.
- We recorded the time of each developer finishing all the refactorings and the names they finally selected for the newly introduced variables.

We leveraged two metrics, i.e., time cost and edit distance, to measure the ability of *VarNamer-C++* to improve the development efficiency. Time cost represents the time developers take to finish all the refactorings. The edit distance (between the names recommended and the names finally selected by developers) reflects developers’ satisfaction with the recommended names. These operations encompass replacing, inserting, and deleting a letter, providing a suitable measure of developers’ editing efficiency.

To avoid the unfairness of this experiment, we took the following measures:

- We ensured that the experience of developers in each group was evenly distributed (the average and median years of development experience for both groups are 2.3 and 2 years, respectively).
- To minimize the interference of irrelevant factors, we completed all necessary preparations before the experiments began. This included opening *Eclipse*, installing CDT and *VarNamer-C++*, opening files, and locating the expressions to be extracted. In addition, all six developers conducted the refactorings on the same personal computer to avoid unnecessary interference. Additionally, developers were unfamiliar with the selected C++ files, and all project information was anonymous.
- Due to the heavy workload of development tasks, developers were instructed to complete each refactoring within 60 seconds to simulate real-world development scenarios.

Table 10. User Study

Metrics	Grouping	Participant A	Participant B	Participant C	Average	Median
Time Cost (seconds)	Group1 (with Eclipse CDT)	1,822	1,850	1,534	1,735	1,822
	Group2 (with VarNamer-C++)	1,431	1,076	1,253	1,253	1,253
Edit Distance	Group1 (with Eclipse CDT)	290	319	309	306	309
	Group2 (with VarNamer-C++)	117	206	141	155	141

Table 11. Results of Prerequisite Condition Inspection on T-test

Metrics	Items	Shapiro-Wilk Test	Levene Test	T-test
Time Cost	statistics	0.99/0.82	0.01	3.35
	p-value	0.99/0.15	0.91	0.03
Edit Distance	statistics	0.97/0.93	1.03	5.42
	p-value	0.66/0.50	0.37	0.01

\*/\*: the statistics or p-value of two groups of data for the confor-mity to normal distribution

The results are presented in Table 10. From this table, we make the following observations:

- Developers in *Group2* finished the task more quickly than those in *Group1*. On average, developers in *Group2* completed the refactorings 482 seconds earlier than those in *Group1*. Using *VarNamer-C++* speeds up extract local variable refactorings by 27.8% compared to CDT, saving 482 seconds out of a total of 1,735 seconds.
- Developers in *Group2* made fewer edits than those in *Group1*. On average, developers in *Group2* made 151 fewer edits than those in *Group1*. Using *VarNamer-C++* reduces edits on recommended variable names by 49.3%, which means 151 fewer edits out of a total of 306 edits.
- Before conducting the t-test analysis on the evaluation metrics, we first conducted Shapiro-Wilk test and Levene test to make sure that the two groups of data (1) conform to a normal distribution and (2) satisfy the homogeneity of variances. The test results are presented in Table 11. The results (i.e., all the p-values are greater than 0.05) suggest that the involved data satisfy the prerequisite condition of the t-test. We then conducted a t-test, which yielded a statistic of 3.35 and a p-value of 0.03 for time cost, and a statistic of 5.42 and a p-value of 0.01 for edit distance. These results indicate significant differences between *Group1* and *Group2* in terms of both metrics.

In conclusion, we underscore the substantial benefits of adopting *VarNamer-C++* in the context of the extract local variable refactorings. Not only does *VarNamer-C++* significantly enhance efficiency by reducing time cost, but it also improves developers' satisfaction by minimizing the need for extra edits of the recommended names. We highlight *VarNamer-C++* as an effective tool for improving the productivity of developers.

*Answer to RQ7:* In terms of time efficiency, *VarNamer-C++* speeds up extract local variable refactorings by 27.8%. In terms of edit efficiency, *VarNamer-C++* reduces edits on recommended variable names by 49.3%.

## 6 DISCUSSION

### 6.1 Threats to Validity

The threat to external validity arises from potential discrepancies between RefactoringMiner’s criteria for identifying extract local variable refactorings and our criteria, which could result in false positives and affect the validity of our experiments. To address this concern, we devised a set of rules to automatically filter out these false positives. These rules were carefully crafted based on the characteristics and patterns of extract local variable refactorings in our dataset. They were designed to identify and exclude instances that did not align with our specific definition of an extract local variable refactoring. By implementing these rules, we aimed to ensure the accuracy and reliability of the refactorings included in our dataset, thus enhancing the validity of our experimental results.

Another potential threat to external validity is that our evaluation of *VarNamer* was limited to a large real-world refactoring dataset for Java, leaving uncertainty about its performance in recommending names for extract local variable refactorings in other programming languages. To mitigate this threat, we took proactive steps to evaluate *VarNamer-C++* on a manually constructed small-scale real-world refactoring dataset specifically tailored for C++. This dataset was carefully curated to include a representative sample of C++ refactorings. We then rigorously assessed the performance of *VarNamer-C++* on this dataset, ensuring that our findings could be generalized beyond the Java context. This approach provides insights into the cross-language applicability and effectiveness of our approach, enhancing the external validity of our study.

The threat to internal validity arises from the potential impact of the parameters used in our study on the performance of *VarNamer*. To address this concern, we conducted parameter tuning on a separate dataset, namely *EmpiricalDataSet*, rather than on the testing dataset *TestingDataset*. This approach ensures that the performance of *VarNamer* remains stable even if changes occur in the testing data.

Additionally, another threat to internal validity stems from the implementation of the IDE baselines, namely *Eclipse* and *IDEA*. The reported performance of these IDEs in our paper may not perfectly mirror their performance in real-world application scenarios. To mitigate this threat, we meticulously implemented *Eclipse* and *IDEA* by invoking internal interfaces and providing all the necessary parameters through the plugin development framework. This approach ensures that the behavior of *Eclipse* and *IDEA* in our experiments closely aligns with their behavior in practical usage scenarios, enhancing the internal validity of our study.

The threat to user study validity stems from the potential influence of different development experiences on the time taken to complete the refactoring task and the preference for certain variable names. To mitigate this threat, we meticulously balanced the years of development experience among participants in each group, ensuring that both the average and median years of development experience were comparable (2.3 and 2 years, respectively). Additionally, we implemented a series of measures outlined in Section 5.10 to guarantee the fairness of the experiment. These measures included providing clear instructions, standardizing the refactoring task, and anonymizing project information to minimize biases. By carefully controlling these factors, we aimed to create a level playing field for all participants, thus enhancing the validity of our user study results. In addition, bias may exist for raters in Section 5.6 since they know the sample is from cases where *VarNamer* outperformed *incoder*. As a result, they might be biased toward finding reasons for this outcome rather than analyzing it more objectively. As a mitigation measure, we had two raters conduct the analysis independently and then discuss any disagreements until a consensus was reached.

## 6.2 Limitations

A major limitation of VarNamer is its current performance, with  $EM_{Precision}$  and  $EM_{Recall}$  metrics of only 41.5% and 33.3%, respectively. These results indicate significant room for improvement, highlighting the challenges associated with automatically suggesting new variable names accurately. An intriguing avenue for future research is to integrate developers' naming preferences into our approach. By leveraging insights into how developers typically name variables, we can potentially enhance *VarNamer*'s performance and address its current limitations more effectively. Incorporating developers' naming preferences could involve analyzing patterns in existing codebases, conducting surveys or interviews with developers to understand their naming conventions, and incorporating this knowledge into the recommendation process.

A second limitation of our approach is its reliance on heuristic rules, which were designed based on the analysis of real-world data. This approach was chosen to ensure that our solution could be seamlessly incorporated into mainstream Integrated Development Environments (IDEs), where efficiency and low latency are paramount. However, while heuristic rules provide a pragmatic solution, they may not capture all nuances of variable naming across different codebases and programming paradigms. Looking ahead, there is an opportunity to leverage the power of advanced Large Language Models (LLMs) and other AI techniques to enhance the variable name recommendation process. By training AI models on vast amounts of code and incorporating contextual understanding, these models have the potential to provide more nuanced and contextually relevant variable name suggestions.

A third limitation of our approach is its current focus on recommending names exclusively for local variables within extract local variable refactorings. While this serves the immediate need for variable name recommendations within this specific refactoring context, it does not extend to recommending names for other identifiers such as method names and class names in related refactorings like extract method and extract class. This limitation stems from the constrained scope of our current implementation, which focuses solely on extract local variable refactorings. However, there is potential for future work to broaden the scope of our approach to encompass a wider range of refactorings that involve the extraction of code entities. By developing mechanisms to recommend names for other types of identifiers, such as methods and classes, our approach could provide more comprehensive support for developers across various refactoring scenarios.

## 7 RELATED WORK

### 7.1 Automatic Variable Renaming

Two notable approaches, Zhang et al.[89] and Liu et al.[62], have been developed specifically to recommend high-quality names for variable renaming tasks. Zhang et al.[89] introduced an identifier renaming prediction and suggestion approach that operates across different granularity levels. Their method begins by predicting whether an identifier requires renaming, then utilizes commit history and naming pattern information to propose a new name. Conversely, *RefBERT*, proposed by Liu et al.[62], employs contrastive learning to recommend names for variables. This approach involves two stages: length prediction and token generation. However, it's important to note that both of these approaches rely on the original name (i.e., the name before refactoring) as one of the inputs to their models. This differs from the task of recommending a name for the extract local variable refactoring, where the objective is to suggest a suitable name for a newly created variable. Therefore, we did not include these approaches as baselines in our study. Additionally, there are several approaches designed to recover variable names for decompiled code, such as DIRE [51], DIRECT [71], and DIRTY [26]. They are designed to handle the variable renaming on deterministically derived representations of stripped binaries. Although it might be possible to pose the variable extract-variable question in a form that can be input to DIRE, DIRECT, or DIRTY, this would require substantial engineering effort and doesn't offer a clear advantage over some advanced code completion techniques, e.g., *Incoder* [36]. Therefore, these approaches were not within the scope of our research.

## 7.2 Code Completion

In addition to the approaches specifically designed for variable renaming tasks, there are also existing code completion tools that have the potential to be applied to the name recommendation process for extract local variable refactorings. These tools, developed by both industry and academia, aim to assist developers by suggesting variables or method calls within an Integrated Development Environment (IDE). One of the pioneering tools in this area is *Prospector*, developed by Mandelin et al. [65]. *Prospector* focuses on suggesting variables or method calls within an IDE environment to enhance developer productivity. Subsequently, a series of tools and plugins have been proposed to further facilitate code completion tasks. These include *InSynth*[43], which provides complete code snippets based on partial input; *Sniff* [24], which offers code suggestions based on context; *IntelliCode*[75], a tool developed by Microsoft that utilizes machine learning to enhance code completion suggestions; *J Sparrow*[8], which provides intelligent code recommendations and automatic code fixes; and *Codata AI Autocomplete* [29], an AI-powered code completion tool that suggests relevant code snippets based on context. These tools leverage various techniques, such as pattern matching, machine learning, and code analysis, to provide accurate and contextually relevant code completion suggestions to developers during software development tasks.

In the academic community, approaches for code completion can be broadly categorized into statistical language model-based and deep learning-based approaches [66]. Statistical language model-based approaches, as demonstrated by Tu et al.[79] and Hellendoorn et al.[45], leverage n-gram models to enhance code completion by incorporating code features. On the other hand, deep learning-based approaches have gained popularity for automated code completion tasks [14, 27, 49]. Kim et al.[49] and Alon et al.[14] proposed methods that integrate syntactic code structures to improve code completion accuracy. Another line of research focuses on automatically renaming variables using deep learning techniques. Liu et al.[60] and Mastropaolet al.[66, 73] pre-trained deep learning language models on large code corpora and fine-tuned them for specific code completion tasks. However, a common limitation of these approaches is their reliance on the context preceding the completion position, which may lead to sub-optimal performance in recommending names for newly extracted variables. This task differs from traditional code completion as it involves completing the code from right to left [36], presenting a challenge for existing models. Consequently, large language models designed for infilling arbitrary code positions have emerged as promising solutions for recommending variable names in extract local variable refactorings. These models treat variable name recommendation as a cloze task and demonstrate efficient performance in this specific task. CodeT5, introduced by Wang et al.[87], is a pre-trained language model with Masked Language Modeling (MLM) as the training objective, making it suitable for code completion tasks. Similarly, UniXcoder[42], a unified multi-modal pre-trained language model, excels in both code understanding and code generation tasks, including code completion. Fried et al. presented Incoder [36], a large generative code model known for its strong performance in infilling arbitrary code regions, achieving competitive results in various code infilling and editing tasks. In addition to these established models, emerging large language models like StarCoder [54], SantaCoder [11], Code Llama [74], OctoCoder (OctoGeeX)[67], and WizardCoder[64] are also gaining attention for their capabilities in code completion tasks. As one of the state-of-the-art models in code completion, Incoder serves as a baseline in our study, representing the latest advancements in deep learning-based approaches.

## 7.3 Improvement of Identifiers' Quality

The quality of identifiers has garnered significant attention due to its profound impact on program comprehension and software maintenance [13, 21, 22, 47, 52, 53, 56, 58, 86]. As a result, various approaches have been developed to enhance the quality of code identifiers, which can be categorized into heuristic-based, statistical language model-based, deep learning-based, information retrieval-based, and generation-based methods. Heuristic-based approaches rely on predefined rules and patterns derived from programming conventions and common coding practices to assess identifier quality and identify inconsistencies. For instance, researchers have proposed heuristic

rules for identifier quality appraisal and the detection of inconsistent identifiers [15, 16, 23, 72, 76]. Statistical language model-based approaches utilize statistical techniques and natural language processing methods to analyze code and improve identifier quality. Allamanis et al.[12] and Lin et al.[57] utilized n-gram language models to identify low-quality identifiers by analyzing code corpus statistics. Deep learning-based approaches leverage neural network architectures to automatically learn representations of code and identify patterns related to identifier quality. While several approaches focus on recommending names for methods [55, 59, 70, 80, 90, 91], there are also those specifically designed to address identifier quality, including variable names. Information retrieval-based approaches, such as those proposed by Liu et al. [63], leverage techniques from information retrieval to recommend names for methods and identify inconsistent method names. Generation-based approaches, similar to those mentioned in DL-based approaches, automatically generate identifier names based on learned representations of code and contextual information. In this paper, we focus on the quality of variable names introduced by extract local variable refactorings, emphasizing the importance of accurately recommending names for newly introduced variables.

## 8 CONCLUSIONS AND FUTURE WORK

Software refactoring is a common practice, and mainstream Integrated Development Environments (IDEs) offer robust tool support for executing refactoring operations. However, existing tool support primarily focuses on the automated execution of predefined refactoring solutions rather than on recommending refactoring opportunities or solutions, particularly regarding the naming of variables. The *Extract Local Variable* refactoring is a prime example where IDEs often fall short in recommending appropriate variable names, despite their proficiency in automatically modifying source code. To address this limitation, we propose *VarNamer*, an automated approach for recommending names for *extract local variable* refactorings by leveraging their contextual information. In this paper, we adopt program static analysis techniques such as lexical analysis, syntax analysis, control flow, and data flow analysis, along with data-mining techniques such as the FP-growth algorithm, to explore real-world refactoring data and design our approach. To evaluate the effectiveness of our proposed approach, we constructed two datasets comprising real-world *extract local variable* refactorings from open-source applications. Our evaluation on these datasets demonstrates that *VarNamer* significantly outperforms state-of-the-art IDEs. Specifically, it improves the chance of exact name matching by 52.6% compared to *Eclipse* and 40.7% compared to *IntelliJ IDEA*. Additionally, a carefully designed user study indicates that our approach accelerates the refactoring process by 27.8% and reduces the need for manual edits by 49.3% on recommended variable names.

Notably, the key heuristic rules of our approach have been merged into *Eclipse* and distributed with its releases. Specifically, we submitted to the *Eclipse* community in total four pull requests that have been approved and merged. Two pull requests [82, 83] implemented the heuristics to suggest new names by reuse, and another two pull requests [81, 84] implemented the heuristics to generate variable names from scratch. In the future, we plan to extend our approach to recommend names for more refactorings like extract method and extract class. It could also be interesting to investigate how to leverage the power of advanced LLMs to further improve the name recommendation.

## ACKNOWLEDGMENTS

The authors would like to say thanks to anonymous reviewers for their insightful comments and suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003, 62172037, and 62141209), China Postdoctoral Science Foundation (No. 2023M740078), and China National Postdoctoral Program for Innovative Talents (BX20240008).

## REFERENCES

- [1] 2024. Eclipse. <http://www.eclipse.org/>
- [2] 2024. Eclipse-CDT. <https://projects.eclipse.org/projects/tools.cdt>
- [3] 2024. Eclipse Java development tools (JDT). <https://www.eclipse.org/jdt/>
- [4] 2024. GitHub. <https://github.com>
- [5] 2024. HuggingFace. <https://huggingface.co>
- [6] 2024. IntelliJ IDEA. <http://www.jetbrains.com/idea/>
- [7] 2024. javalang. <https://github.com/c2nes/javalang>
- [8] 2024. JSparrow. <https://jsparrow.io/>
- [9] 2024. NetBeans. <http://netbeans.org/>
- [10] 2024. Visual Studio. <https://visualstudio.com/>
- [11] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).
- [12] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 281–293.
- [13] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings* (2015), 38–49. <https://doi.org/10.1145/2786805.2786849>
- [14] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International conference on machine learning*. PMLR, 245–256.
- [15] Reem Alsuhaihani, Christian Newman, Michael Decker, Michael Collard, and Jonathan Maletic. 2021. On the naming of methods: A survey of professional developers. *Proceedings - International Conference on Software Engineering* (2021), 587–599. <https://doi.org/10.1109/ICSE43902.2021.00061>
- [16] Reem S. Alsuhaihani, Christian D. Newman, Michael John Decker, Michael L. Collard, and Jonathan I. Maletic. 2022. An approach to automatically assess method names. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16–17, 2022*, Ayushi Rastogi, Rosalia Tufano, Gabriele Bavota, Venera Arnaoudova, and Sonia Haiduc (Eds.). ACM, 202–213. <https://doi.org/10.1145/3524610.3527780>
- [17] Apache. 2024. *Apache/arrow*. Retrieved July 23, 2024 from <https://github.com/apache/arrow>
- [18] Apache. 2024. *Apache/mesos*. Retrieved July 23, 2024 from <https://github.com/apache/mesos>
- [19] Apache. 2024. *FalsePositiveExample*. Retrieved July 23, 2024 from <https://github.com/apache/cassandra/commit/d292327#diff-b9a8560c03d1f86c3104c171da868f801af104fb2d76ce4b7d7eec627403309a>
- [20] Apache. 2024. *RefactoringExample1*. Retrieved July 23, 2024 from <https://github.com/apache/camel/commit/6ad3c0e#diff-62a7cc451d91c504fd95c944a900dadce9c298a4e37a0ef2b38d09e492e379b9>
- [21] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 31–35.
- [22] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 156–165.
- [23] Caprile and Tonella. 2000. Restructuring program identifier names. In *Proceedings 2000 International Conference on Software Maintenance*. IEEE, 97–107.
- [24] Shaunik Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. Sniff: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering: 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22–29, 2009. Proceedings 12*. Springer, 385–400.
- [25] checkstyle. 2024. *RefactoringExample4*. Retrieved July 23, 2024 from <https://github.com/checkstyle/checkstyle/commit/5b45d5a#diff-a16f333e36f0ee0bfdcc4164a70c6977c34dc574c37c4c7ed79718712906d30c>
- [26] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*. 4327–4343.
- [27] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, et al. 2024. Deep Learning-based Software Engineering: Progress, Challenges, and Opportunities. *arXiv preprint arXiv:2410.13110* (2024).
- [28] Xiaye Chi, Hui Liu, Guangjie Li, Weixiao Wang, Yunni Xia, Yanjie Jiang, Yuxia Zhang, and Weixing Ji. 2023. An Automated Approach to Extracting Local Variables. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3–9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 313–325. <https://doi.org/10.1145/3611643.3616261>
- [29] Codota. 2024. *Codota AI Autocomplete for Java and JavaScript in IntelliJ*. Retrieved July 23, 2024 from <https://plugins.jetbrains.com/plugin/7638-codota-ai-autocomplete-for-java-and-javascript>

- [30] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
- [31] Florian Deissenboeck and Markus Pizka. 2006. Concise and consistent naming. *Software Quality Journal* 14, 3 (2006), 261–282. <https://doi.org/10.1007/s11219-006-9219-1>
- [32] dubbo. 2024. *RefactoringExample2*. Retrieved July 23, 2024 from <https://github.com/dubbo/dubbo/commit/2eaa132#diff-6d63b3c410d6fad6b8661d7faa486639f7f6ac939f7f8f7732f6a33d149b036a>
- [33] Liu et al. 2024. *JavaRepos*. Retrieved July 23, 2024 from <https://github.com/TruX-DTF/debug-method-name/tree/master/Data/JavaRepos/>
- [34] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 313–324.
- [35] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering* 33, 11 (2007), 725–743.
- [36] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. <https://openreview.net/pdf?id=hQwb-lbM6EL>
- [37] geoserver. 2024. *RefactoringExample3*. Retrieved July 23, 2024 from <https://github.com/geoserver/geoserver/commit/c54d9cc#diff-40dd1af56b59d542f9c41036b47c0db77b2b4a5bc8ece7a2f4a78cc196867181>
- [38] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1303–1313.
- [39] Google. 2024. *Google/angle*. Retrieved July 23, 2024 from <https://github.com/google/angle>
- [40] Google. 2024. *Google/dawn*. Retrieved July 23, 2024 from <https://github.com/google/dawn>
- [41] Google. 2024. *Google/skia*. Retrieved July 23, 2024 from <https://github.com/google/skia>
- [42] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).
- [43] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16–19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 27–38. <https://doi.org/10.1145/2491956.2462192>
- [44] Jiawei Han and Jian Pei. 2000. Mining frequent patterns by pattern-growth. *ACM SIGKDD Explorations Newsletter* 2, 2 (2000), 14–20. <https://doi.org/10.1145/380995.381002>
- [45] 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.
- [46] Matthew Honnibal and Ines Montani. 2024. spaCy. <https://spacy.io/>
- [47] Einar W Høst and Bjarte M Østvold. 2009. Debugging method names. In *European Conference on Object-Oriented Programming*. Springer, 294–317.
- [48] JetBrains. 2024. *Extract/Introduce variable*. Retrieved July 23, 2024 from <https://www.jetbrains.com/help/idea/extract-variable.html>
- [49] 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.
- [50] Sotiris Kotsiantis and Dimitris Kanellopoulos. 2006. Association Rules Mining: A Recent Overview. *Science* 32, 1 (2006), 71–82.
- [51] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 628–639.
- [52] Dawn Lawrie, Henry Feild, and David Binkley. 2007. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering* 12 (2007), 359–388.
- [53] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What’s in a name? A study of identifiers. *IEEE International Conference on Program Comprehension 2006* (2006), 3–12. <https://doi.org/10.1109/ICPC.2006.51>
- [54] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Lucioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! (2023), 1–54. [arXiv:2305.06161](https://arxiv.org/abs/2305.06161) <http://arxiv.org/abs/2305.06161>
- [55] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22–30 May 2021*. IEEE, 574–586.

- <https://doi.org/10.1109/ICSE43902.2021.00060>
- [56] B. Lin, C. Nagy, G. Bavota, A. Marcus, and M. Lanza. 2019. On the Quality of Identifiers in Test Code. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 204–215. <https://doi.org/10.1109/SCAM.2019.00031>
- [57] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza. 2017. Investigating the Use of Code Analysis and NLP to Promote a Consistent Usage of Identifiers. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 81–90. <https://doi.org/10.1109/SCAM.2017.17>
- [58] Bo Liu, Hui Liu, Nan Niu, Yuxia Zhang, Guangjie Li, and Yanjie Jiang. 2023. Automated Software Entity Matching Between Successive Versions. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE, 1615–1627. <https://doi.org/10.1109/ASE56229.2023.00132>
- [59] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to recommend method names with global context. In *Proceedings of the 44th International Conference on Software Engineering*. 1294–1306.
- [60] 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.
- [61] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen est Omen: Exploring and Exploiting Similarities between Argument and Parameter Names. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1063–1073. <https://doi.org/10.1145/2884781.2884841>
- [62] Hao Liu, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023. RefBERT: A Two-Stage Pre-trained Framework for Automatic Rename Refactoring. (2023). <https://doi.org/10.1145/3597926.3598092> arXiv:2305.17708
- [63] Kui Liu, Dongsun Kim, Tegawende F. Bissyande, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to Spot and Refactor Inconsistent Method Names. *Proceedings - International Conference on Software Engineering 2019-May (2019)*, 1–12. <https://doi.org/10.1109/ICSE.2019.00019>
- [64] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Dixin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).
- [65] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. *ACM Sigplan Notices* 40, 6 (2005), 48–61.
- [66] Antonio Mastropaoletti, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2023. Automated variable renaming: are we there yet? *Empirical Software Engineering* 28, 2 (2023), 1–26. <https://doi.org/10.1007/s10664-022-10274-8> arXiv:2212.05738
- [67] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Bin Yuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124* (2023).
- [68] Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Trans. Software Eng.* 38, 1 (2012), 5–18. <https://doi.org/10.1109/TSE.2011.41>
- [69] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactorings. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7920 LNCS* (2013), 552–576. [https://doi.org/10.1007/978-3-642-39038-8\\_23](https://doi.org/10.1007/978-3-642-39038-8_23)
- [70] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1372–1384. <https://doi.org/10.1145/3377811.3380926>
- [71] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. DIRECT: A Transformer-based Model for Decompiled Variable Name Recovery. *NLP4Prog 2021* (2021), 48.
- [72] Anthony Peruma, Venera Arnaoudova, and Christian D. Newman. 2021. IDEAL: An Open-Source Identifier Name Appraisal Tool. (2021). arXiv:2107.08344 <http://arxiv.org/abs/2107.08344>
- [73] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* 21 (2020), 1–67. arXiv:1910.10683
- [74] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [75] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofiqi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 329–340.
- [76] Andreas Thies and Christian Roth. 2010. Recommending rename refactorings. *Proceedings - International Conference on Software Engineering* (2010), 1–5. <https://doi.org/10.1145/1808920.1808921>
- [77] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Trans. Software Eng.* 48, 3 (2022), 930–950. <https://doi.org/10.1109/TSE.2020.3007722>
- [78] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 6–13, 2018*. IEEE, 103–114.

- Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 483–494. <https://doi.org/10.1145/3180155.3180206>
- [79] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 269–280. <https://doi.org/10.1145/2635868.2635875>
- [80] 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*. 741–753.
- [81] Taiming Wang. 2024. *Automated Name Recommendation For The Extract Local Variable Refactoring*. Retrieved July 23, 2024 from <https://github.com/eclipse-jdt/eclipse.jdt.ui/pull/602>
- [82] Taiming Wang. 2024. *Context-based-name-recommendation*. Retrieved July 23, 2024 from <https://github.com/eclipse-jdt/eclipse.jdt.ui/pull/656>
- [83] Taiming Wang. 2024. *Extract Similar Expression in All Methods If End-Users Want*. Retrieved July 23, 2024 from <https://github.com/eclipse-jdt/eclipse.jdt.ui/pull/680>
- [84] Taiming Wang. 2024. *Recommend variable name for Extracted Local Variable Refactoring when the extracted expression is a method invocation*. Retrieved July 23, 2024 from <https://github.com/eclipse-jdt/eclipse.jdt.ui/pull/685>
- [85] Taiming Wang. 2024. *VarNamer*. Retrieved July 23, 2024 from <https://github.com/Michaelll123/VarNamer>
- [86] Taiming Wang, Yuxia Zhang, Lin Jiang, Yi Tang, Guangjie Li, and Hui Liu. 2025. Deep learning based identification of inconsistent method names: How far are we? *Empirical Software Engineering* 30, 1 (2025), 31.
- [87] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [88] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence* 29, 6 (2007), 1091–1095.
- [89] Jingxuan Zhang, Junpeng Luo, Jiahui Liang, Lina Gong, and Zhiqiu Huang. 2023. An Accurate Identifier Renaming Prediction and Suggestion Approach. *ACM Transactions on Software Engineering and Methodology* (2023). <https://doi.org/10.1145/3603109>
- [90] Jie Zhu, Lingwei Li, Li Yang, Xiaoxiao Ma, and Chun Zuo. 2023. Automating Method Naming with Context-Aware Prompt-Tuning. *arXiv preprint arXiv:2303.05771* (2023).
- [91] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318* (2021).