Investigating the Use of Code Analysis and NLP to Promote a Consistent Usage of Identifiers

Bin Lin*, Simone Scalabrino†, Andrea Mocci*, Rocco Oliveto†, Gabriele Bavota*, and Michele Lanza*
**Università della Svizzera italiana (USI), Switzerland* — †University of Molise, Italy

***Abstract*—Meaningless identifiers as well as inconsistent use of identifiers in the source code might hinder code readability and result in increased software maintenance efforts. Over the past years, effort has been devoted to promoting a consistent usage of identifiers across different parts of a system through approaches exploiting static code analysis and Natural Language Processing (NLP). These techniques have been evaluated in small-scale studies, but it is unclear how they compare to each other and how they complement each other. Furthermore, a full-fledged larger empirical evaluation is still missing.**

We aim at bridging this gap. We asked developers of five projects to assess the meaningfulness of the recommendations generated by three techniques, two already existing in the literature (one exploiting static analysis, one using NLP) and a novel one we propose. With a total of 922 rename refactorings evaluated, this is, to the best of our knowledge, the largest empirical study conducted to assess and compare rename refactoring tools promoting a consistent use of identifiers. Our study sheds light on the current state-of-the-art in rename refactoring recommenders, and indicates directions for future work.

I. Introduction

In programming languages, identifiers are used to name program entities; *e.g.,* in Java, identifiers include names of packages, classes, interfaces, methods, and variables. Identifiers account for ∼30% of the tokens and ∼70% of the characters in the source code [1]. Naming identifiers in a careful, meaningful, and consistent manner likely eases program comprehension and supports developers in building consistent and coherent conceptual models [2].

Instead, poorly chosen identifiers might create a mismatch between the developers' cognitive model and the intended meaning of the identifiers, thus ultimately increasing the risk of fault proneness. Indeed, several studies have shown that bugs are more likely to reside in code with low quality identifiers [3], [4]. Arnaoudova *et al.* [5] also found that methods containing identifiers with higher physical and conceptual dispersion are more fault-prone. This suggests the important role played by a specific class of identifiers, *i.e.,* local variables and method parameters, in determining the quality of methods.

Naming conventions can help to improve the quality of identifiers. However, they are often too general, and cannot be automatically enforced to ensure consistent and meaningful identifiers. For example, the Java Language Specification[1] indicates rules for naming local variables and parameters: *e.g., "should be short, yet meaningful"*. Clearly, these requirements do not guarantee consistent variable naming.

For example, developers might use "`localVar`" and "`varLocal`" in different code locations even if these two names are used in the same context and with the same meaning. Also, synonyms might be used to name the same objects, such as "`car`" and "`auto`". Finally, developers might not completely adhere to the rules defined in project-specific naming conventions.

Researchers have presented tools to support developers in the consistent use of identifiers. Thies and Roth [6] analyzed variable assignments to identify pairs of variables likely referring to the same object but named differently. Allamanis *et al.* [7] pioneered the use of NLP techniques to support identifiers renaming. Their NATURALIZE tool exploits a language model to infer from a code base the naming conventions and to spot *unnatural* identifiers (*i.e.,* unexpected identifiers), that should be renamed to promote consistency.

To obtain a reliable evaluation of approaches supporting automatic identifier renaming, the original authors of the source code should be involved in assessing the meaningfulness of the suggested refactorings. However, running such evaluations is expensive, thus refactoring techniques are often evaluated in "artificial scenarios" (*e.g.,* injecting a meaningless identifier in the code and check whether the tool is able to recommend a rename refactoring for it) and/or by relying on the manual evaluation of a limited number of recommended rename refactorings. For example, Thies and Roth [6] manually assessed the meaningfulness of 32 recommendations generated by their tool. Instead, Allamanis *et al.* [7] firstly analyzed 33 rename recommendations generated by NATURALIZE, and then opened pull requests in open source projects to evaluatethe meaningfulness of 18 renaming recommended by NATURALIZE (for a total of 51 data points).

We aim at assessing the meaningfulness of the rename refactorings recommended by state-of-the-art approaches on a larger scale (922 evaluations in total) and by only relying on developers having a first-hand experience on the object systems of our study. We evaluated two existing approaches, *i.e.,* the one by Thies and Roth [6] exploiting static code analysis, and the NATURALIZE tool [7] using NLP techniques to support identifier renaming. In addition, we propose a variation of NATURALIZE, named LEAR (LExicAl Renaming), exploiting a different concept of language model more focused on the lexical information present in the code. We conducted extensive empirical comparison of these three tools. Our results support the potential practical use of the identifier renaming approaches and indicates directions for improvement.

[1] https://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html

II. Related Work

We discuss the literature related to the study of identifiers' quality and to techniques supporting the automatic identifier renaming. We describe in detail two of the techniques that are part of our empirical study, and in particular the approach by Thies and Roth [6] and the NATURALIZE tool by Allamanis *et al.* [7]. The third approach involved in our evaluation, named LEAR, is described in Section III.

Lawrie *et al.* [8] report the results of an experiment in which over 100 developers were asked to describe 12 different functions. The functions used three different types of identifiers, *i.e.,* single letters, abbreviations, and full words. The results showed that developers tend to comprehend identifiers composed of full words better than single letters/abbreviations. Lawrie *et al.* [9] also investigated the identifier quality based on almost 50 million lines of code, covering different programming languages. They found that modern software projects have better quality of identifier names than old projects.

Butler *et al.* [3] used eleven identifier naming guidelines for Java to evaluate the quality of identifiers. They found statistically significant associations between the identifier names violating at least one guideline and code quality issues reported by a static analysis tool. Based on this finding, Butler *et al.* [10] conclude that some of these naming guidelines can be used as a light-weight diagnostic to identify areas of potentially problematic code. Murphy-Hill *et al.* [11] investigated the adoption of refactoring tools in the IDE, reporting that *rename refactoring* is among the most frequently performed operations.

Much effort has been devoted to improving the quality of identifier names, for example via identifier splitting [12], [13], [14] and expansion [15], [16]. However, these approaches cannot address the problem caused by non-adherence to naming conventions or by the inconsistent use of identifiers.

Reiss [17] proposed a tool that learns code style from existing source code, such as identifier conventions and indentation, and applies it automatically on a new code artifact, thus making it consistent with the rest of the system. A similar tool is SmartFormatter [18] that also learns the lexical form of terms used in identifiers.

Caprile and Tonella [19] proposed an approach to restructure identifiers with the goal of enhancing their meaningfulness. The approach builds a standard lexicon dictionary and a synonyms dictionary by analyzing a set of programs. Then, when analyzing a new program, the devised approach decomposes each identifier into the terms composing it and checks whether each term is "standard" according to the built dictionary. Non-standard terms are suggested to be replaced by their standard forms (*e.g.,* expand `upd` into `update`). While their approach was foundational for the field of identifier restructuring, we do not consider it in our empirical study since we focus on techniques aimed at promoting a consistent use of identifiers across a system. The approach by Caprile and Tonella [19] is focused on improving the meaningfulness of identifiers, without considering their consistent use.

Høst and Østvold [20] presented an approach to identify *naming bugs*, *i.e.,* a method name not representative of its implementation. The approach mines method naming rules from a corpus of Java applications, and suggests renamings for methods not following the learned rules. In our study we did not consider the approach by Høst and Østvold since we focus on techniques recommending identifier renames for methods' variables and parameters. Feldthaus and Møller [21] proposed a technique to support rename refactorings in JavaScript. When a developer decides to rename a variable v, a static analysis technique is applied to identify v's occurrences that need to be consistently renamed. The list of identified occurrences is provided to the developer for inspection. Jablonski and Hou [22] proposed CReN, a tool to track copy-and-paste clones and support identifier renaming in the IDE. A set of rules based on relationships between identifiers is used to infer developers' intentions (*e.g.,* two identifiers that are frequently renamed together). Also these two approaches have not been considered in our study since we focus on techniques suggesting renaming operations to promote a consistent use of identifiers.

A. Thies and Roth [6] - Static code analysis

Thies and Roth [6] present a tool to support identifier renaming based on information extracted via static code analysis. The main idea is to exploit information derived by variable assignments to identify the inconsistent use of identifiers to name variables referring to the same object. The authors consider two types of assignments: 1) a variable is assigned to another variable (*e.g.,* `paper` = `bestPaper`); 2) a variable is assigned to a method invocation (*e.g.,* `paper` = `getBestPaper`()). For the second case, the assignment can be seen as the assignment to the variable returned by the method. In our example, assume that the method `getBestPaper()` returns a variable named "`bestPaper`", the assignment is treated as `paper` = `bestPaper`. Once the information about variable assignments is extracted for all variables, an assignment graph is constructed where each node represents a variable and an edge connecting two variables represents an assignment. If an edge connects two nodes named by using different identifiers but representing two variables of the same type, the tool generates a rename recommendation.

To evaluate their approach, Thies and Roth [6] applied their tool to four open source projects, and manually inspected renaming suggestions generated for variables with non-primitive types. As a result, 21 out of 32 suggestions appear to be beneficial. Among the 21 useful suggestions, 4 of them are related to synonyms and 17 to inaccurate choice of the identifiers.

In our study, we re-implemented the approach by Thies and Roth since their tool is not publicly available and we refer to this approach as CA-RENAMING, to stress the fact that it only relies on static **c**ode **a**nalysis. We selected this approach because it is one of the very few existing approaches to reduce the inconsistent use of identifiers, while most approaches focus on increasing the meaningfulness of identifiers without considering naming consistency.

B. Allamanis et al. *[7] - NLP*

Allamanis *et al.* [7] present a framework, named NATURALIZE, to recommend natural identifier names and formatting conventions by applying NLP to source code. One of the goals of NATURALIZE is to promote identifier consistency. NATURALIZE exploits a n-gram language model to estimate the probability that a specific identifier should be used in a given context to name a variable. Language models are widely employed in many domains such as speech recognition and code completion. The n-gram model is one of the most commonly used language models and it determines the probability of having a word w_i given the previous n-1 words. This probability is denoted by $p(w_i|w_{i-1}, w_{i-2}, \dots, w_{i-n+1})$, where $w_{i-n+1}, \dots, w_{i-1}, w_i$ are n continuous words. The probability that w_i follows $w_{i-n+1}, \dots, w_{i-2}, w_{i-1}$ is estimated by training the language model on a training set, composed of textual documents. When applying the language model to software-related tasks, like code completion, the training set is composed of code documents.

NATURALIZE follows a two-step approach to recommend a rename refactoring for a variable v:

1) **Generating candidate names**. NATURALIZE uses the AST of the program under analysis to find the set of locations, L, in which v appears. Then, it builds a snippet S representing the *context* in which v is used by taking the lowest common ancestor in AST of nodes in L [7]. S is then linearly scanned by using a moving window of length n, where n is the number of tokens. A token could be an identifier, a syntactic symbol of the programming language, like ";", a reserved keyword of the language and so on. All n-grams containing v are extracted and the collection of these n-grams becomes the context set of v. If another variable v_i other than v occurs in at least one similar context (*i.e.,* in at least one similar n-gram), a new snippet S_i is created, *i.e., S* with all v replaced by v_i, and it is added to the list of alternative candidates.
2) **Ranking candidates**. A score function leveraging a language model is defined to rank the candidates generated in the previous step. While any probability model can be used in the score function, the authors apply the n-gram language model we previously describe to assess the probability of a given candidate. In other words, given the context (*i.e.,* the set of n-grams) where an identifier v is used, the probability of renaming v into v_i is higher if v_i is used in the training set in which the language model has been built in a similar context.

To evaluate NATURALIZE, the authors assessed the meaningfulness of the refactorings recommended for 30 methods (for a total of 33 variable renamings). Half of the suggestions were identified as meaningful. Also, they submitted 18 patches to five GitHub projects, among which 14 were accepted.

The goal of NATURALIZE (*i.e.,* promoting consistency), its peculiarity of relying on NLP techniques, and its availability[2], made it an obvious choice for our study.

We started from the core idea behind NATURALIZE (*i.e.,* using a language model to promote a consistent use of identifiers) to define an alternative rename refactoring approach, named LEAR, that is presented in the next section and tries to overcome a number of possible limitations of the NATURALIZE approach. For example, NATURALIZE uses all textual tokens in the n-gram language model (including, *e.g.,* punctuation) to characterize the context in which an identifier is used; we believe that all the *syntactic sugar* in the programming language could mostly represent noise for the language model, thus reducing the quality of the rename recommendations. Also, NATURALIZE does not verifies whether the recommended rename refactorings are valid or not (*e.g.,* it is not possible to rename an identifier *id* used in id_s in a method m, if id_s is already used in m to name any other variable/parameter. We present LEAR in the next section, by paying particular attention to stressing its main differences with respect to NATURALIZE.

III. LExicAL Renaming

Our LEAR recommends renaming operations related to (i) variables declared in methods and (ii) method parameters. The renaming of methods/classes as well as of instance/class variables is not currently supported, since, as it will be clearer later, LEAR works at method level. The support of other types of identifiers is part of our future work agenda. In the following we describe in detail the main steps of LEAR.

Identifying methods and extracting the vocabulary. LEAR parses the source code of the input system by relying on the `srcML` infrastructure [23]. The goal of the parsing is to extract (i) the complete list of methods, and (ii) the identifiers' vocabulary, defined as the list of all the identifiers used to name parameters and variables (declared at both method and class level) in the whole project. From now on we refer to the identifiers' vocabulary simply as the *vocabulary*. Once the vocabulary and the list of methods have been extracted, the following steps are performed for each method m in the system. We use the method in Listing 1 as a running example.

N-gram Extraction from m. We extract all textual tokens from the method m under analysis, by removing (i) comments and string literals, (ii) all non-textual content, *i.e.,* punctuation and (iii) non-interesting words, such as Java keywords and the name of method m itself. Basically, we only keep tokens referring to identifiers, excluding the name of m, and non-primitive types, which are Java keywords. This is one of the main differences with respect to NATURALIZE.

Indeed, while NATURALIZE uses all textual tokens in the n-gram language model (including, *e.g.,* Java keywords), we only focus on tokens containing *lexical* information. We expect sequences of only lexical tokens to better capture and characterize the context in which a given identifier is used.

Listing 1: Example of method analyzed

```
public void printUser(int uid) {
    String q = "SELECT␣*␣WHERE␣user_id␣=␣" + uid;
    User user = runQuery(q);
    System.out.println(user);
}
```

[2] http://groups.inf.ed.ac.uk/naturalize/

The list of identifiers extracted from `printUser` includes: `uid`, `String`, `q`, `uid`, `User`, `user`, `runQuery`, `q`, `System`, `out`, `println`, `user`. Again, our conjecture is that such a list of tokens captures the *context*—referred to method `printUser`—where an identifier (*e.g.,* `q`) is used. After obtaining the identifier list, we extract n-grams from it such that the language model can use them to estimate the probability that a specific identifier should be used in a given context.

Lin *et al.* [24] found that the n-gram language model achieves the best accuracy in supporting code completion tasks when setting $n = 3$. The same value was used in the original work by Hindle *et al.* [25] proposing the usage of the language model for code completion. Therefore, we build 3-grams from the extracted list of tokens. In our running example, ten 3-grams will be extracted, including: ⟨`uid`, `String`, `q`⟩, ⟨`String`, `q`, `uid`⟩, ⟨`q`, `uid`, `User`⟩, ⟨`uid`, `User`, `user`⟩, *etc.*

Generating candidate rename refactoring. For each variable/parameter identifier in m (in the case of `printUser`: `uid`, `q`, and `user`), LEAR looks for its possible renaming by exploiting the *vocabulary* built in the first step. Given an identifier under analysis id, LEAR extracts from the *vocabulary* all the identifiers id_s which meet the following constraints:

- C_1: id_s is used to name a variable/parameter of the same type as the one referred by id. For example, if id is a parameter of type `int`, id_s must be used at least once as an `int` variable/parameter;
- C_2: id_s is not used in m to name any other variables/-parameters. Indeed, in such a circumstance, it would not be possible to rename id in id_s in any case;
- C_3: id_s is not used to name any attribute of the class C_k implementing m nor in any class C_k extends, for the same reason explained in C_2.

The constraint checking not considered in NATURALIZE represents another difference between LEAR and NATURALIZE.

We refer to the list of valid identifiers fulfilling the above criteria as VI_{id}. Then, LEAR uses a customized version of the 3-gram language model to compute the probability that each identifier id_s in VI_{id} appears, instead of id, in all the 3-grams of m including id.

Let TP_{id} be the set of *3-gram patterns* containing at least once id, and $tp_{id \to id_s}$ be a 3-gram obtained from a pattern $tp_{id} \in TP_{id}$ where the variable id is replaced with a valid identifier $id_s \in VI_{id}$. We define the probability of a given substitution to a variable as:

$$P(tp_{id \to id_s}) = \frac{count(tp_{id \to id_s})}{\sum_{y \in VI_{id}} count(tp_{id \to y})}$$

When the pattern is in the form of $\langle id_1, id_2, id \rangle$, the probability of a substitution corresponds to the classic probability as computed by a 3-gram language model, that is:

$$\begin{aligned} P(\langle id_1, id_2, id \rangle_{id \to id_s}) &= P(id_s | id_1, id_2) \\ &= \frac{count(\langle id_1, id_2, id_s \rangle)}{count(\langle id_1, id_2 \rangle)} \end{aligned}$$

To better understand this core step of LEAR, let us discuss what happens in our running example when LEAR looks for possible renaming of the `uid` parameter identifier. The 3-grams of `printUser` containing `uid` are: ⟨`uid`, `String`, `q`⟩, ⟨`String`, `q`, `uid`⟩, ⟨`q`, `uid`, `User`⟩, and ⟨`uid`, `User`, `user`⟩.

Assume that the list of identifiers VI_{id} (*i.e.,* the list of valid alternative identifiers for `uid`) includes `userId` and `localCount`. LEAR uses the language model to compute the probability that `userId` occurs in each of the 3-grams of `printUser` containing `uid`. For example, the probability of observing `userId` in the 3-gram ⟨`q`, `uid`, `User`⟩ is:

$$p(\texttt{q}, \texttt{userId}, \texttt{User}) = \frac{count(\texttt{q}, \texttt{userId}, \texttt{User})}{count(\texttt{q}, \texttt{y}, \texttt{User})}$$

where $count(\texttt{q}, \texttt{userId}, \texttt{User})$ is the number of occurrences of the 3-gram $\langle \texttt{q}, \texttt{userId}, \texttt{User} \rangle$ in the system, and $count(\texttt{q}, \texttt{y}, \texttt{User})$ is the number of occurrences of the corresponding 3-gram, where `y` represents any possible identifier (including `userId` itself). Note that the *count* function only considers n-grams where id_s has the same type as id. Also, it does not take into account n-grams extracted from the method under analysis. This is done to avoid favoring the probability of the current identifier name used in the method under analysis as compared to the probability of other identifiers.

How the probability for a given identifier to appear in a n-gram is computed also differentiates LEAR from NATURALIZE. In the example reported above, NATURALIZE in fact computes the probability of observing `User` following $\langle \texttt{q}, \texttt{userId} \rangle$:

$$p(\texttt{User} | \texttt{q}, \texttt{userId}) = \frac{count(\texttt{q}, \texttt{userId}, \texttt{User})}{count(\texttt{q}, \texttt{userId})}$$

The two probabilities (*i.e.,* the one computed by LEAR and by NATURALIZE), while based on similar intuitions, could clearly differ. Our probability function is adapted from the standard language model (*i.e.,* the one used by NATURALIZE) in an attempt to better capture the context in which an identifier is used. This can be noticed in the way our denominator is defined: it keeps intact that identifiers' context in which we are considering injecting `userId` instead of `uid`.

The average probability across all these 3-grams is considered as the probability of id_s being used instead of id in m.

This process results in a ranked list of VI_{id} identifiers having on top the identifier with the highest average probability of appearing in all the 3-grams of m as a replacement (*i.e.,* rename) of id. We refer to this top-ranked identifier as T_{id}.

Finally, LEAR uses the same procedure to compute the average probability that the identifier id itself appears in the 3-grams where it currently is. If the T_{id} has the higher probability of appearing in the 3-grams is than id, a candidate rename refactoring has been found (*i.e.,* rename id in T_{id}). Otherwise, no rename refactoring is needed.

Assessing the confidence and the reliability of the candidate recommendations. LEAR uses two indicators acting as proxies for the *confidence* and the *reliability* of the recommended refactoring. Given a rename refactoring recommendation $id \to T_{id}$ in the method m, the confidence indicator is the average probability of T_{id} to occur instead of id in the 3-grams of m where id appears.

We refer to this indicator as C_p, and it is defined in the [0, 1] interval. The higher C_p, the higher the confidence of the recommendation. We study how C_p influences the quality of the recommendations generated by LEAR in the following.

The "reliability" indicator, named C_c, is the number of distinct 3-grams used by the language model in the computation of C_p for a given recommendation $id \to T_{id}$ in the method m. Given $\langle id_1, id_2, id \rangle$ a 3-gram where id appears in m, we count the number of 3-grams in the system in the form $\langle id_1, id_2, x \rangle$, where x can be any possible identifier. This is done for all the 3-grams of m including id, and the sum of all computed values is represented by C_c. The conjecture is that the higher C_c, the higher is the reliability of the C_p computation. Indeed, the higher C_c, the higher the number of 3-grams from which the language model learned that T_{id} is a good substitution for id. C_c is unbounded on top. We study what is the minimum value of C_c allowing reliable recommendations in the following.

Note that while NATURALIZE does also provide a scoring function based on the probability derived by the n-gram language model to indicate the confidence of the recommendation (*i.e.,* the equivalent of our C_p indicator), it does not implement a "reliability" indicator corresponding to C_c.

Tuning of the C_c and C_p indicators. To assess the influence of the C_p (confidence) and C_c (reliability) indicators on the quality of the rename refactorings generated by LEAR, we conducted a study on one system, named SMOS. We asked one of the SMOS developers (having nowadays six years of industrial experience) to assess the meaningfulness of the LEAR recommendations. SMOS is a Java web application developed by a team of Master students, and composed by 121 classes for a total of ~23 KLOC. We used the SMOS system only for the tuning of the indicators C_p and C_c, *i.e.,* to identify minimum values needed to receive meaningful recommendations for both of them. SMOS is not used in the actual evaluation of our approach, presented in Section IV.

We ran LEAR on the whole system and asked the participant to analyze the 146 rename refactoring generated by LEAR and to answer, for each of them, the question *Would you apply the proposed refactoring?*, assigning a score on a three-point Likert scale: 1 (yes), 2 (maybe), and 3 (no). We clarified with the participant the meaning of the three possible answers:

1 (yes) must be interpreted as "*the recommended renaming is meaningful and should be applied*", *i.e.,* the recommended identifier name is better than the current one;

2 (maybe) must be interpreted as "*the recommended renaming is meaningful, but should not be applied*", *i.e.,* the recommended identifier is a valid alternative to the one currently used, but is not a better choice;

3 (no) must be interpreted as "*the recommended rename refactoring is not meaningful*".

The participant answered *yes* to 18 (12%) of the recommended refactoring, *maybe* to 15, and *no* to 113. This negative trend is expected, considering the fact that we asked the participant to assess the quality of the recommended refactoring independently from the values of the C_p and the C_c indicators. That is, given the goal of this study, also recommendations having very low values for both indicators (*e.g.,* $C_p = 0.1$ and $C_c = 1$) were inspected, despite we do not expect them to be meaningful. Table I reports five representative examples of rename refactoring tagged with a *yes* by the developer.

TABLE I: Five rename refactoring tagged with a *yes*

Original name	**Rename**	C_p	C_c
mg	managerUser	1.00	146
e	invalidValueException	0.90	356
buf	searchBuffer	0.89	5
result	classroom	0.87	15
managercourseOfStudy	managerCourseOfStudy	0.67	12

By inspecting the assessment performed by the participant, the first thing we noticed is that recommendations having $C_c < 5$ (*i.e.,* less than five distinct 3-grams have been used by the language model to learn the recommended rename refactoring) are generally unreliable, and should not be considered. Indeed, out of the 28 rename refactoring having $C_c < 5$, one (3%) was accepted (answer "*yes*") by the developer and three (10%) were classified as *maybe*, despite the fact that 22 of them had $C_p = 1.0$ (*i.e.,* the highest possible confidence for the generated recommendation). Thus, when $C_c < 5$ even recommendations having a very high confidence are simply not reliable. When $C_c \geq 5$, we noticed that its influence on the quality of the recommended renames is limited, *i.e.,* no other clear trend in the quality of the recommended refactoring can be observed for different values of C_c. Thus, we excluded the 28 refactoring recommendations having $C_c < 5$ and studied the role played by C_p in the remaining 118 recommendations (17 *yes*, 12 *maybe*, and 89 *no*).

Fig. 1 reports the recall and precision levels of our approach when excluding the recommendations having $C_p < t$, with t varying between 1.0 and 0.1 at steps of 0.1. Note that in the computation of the recall and precision we considered the 29 recommendations accepted with a *yes* (17) or assessed as meaningful with a *maybe* (12) as correct (*i.e.,* the *maybe* answers are equated to the *yes* answers, and considered correct). This choice was dictated by the fact that we see the meaningful recommendations tagged with *maybe* as valuable for the developer, since she can then decide whether the alternative identifier name provided by our approach is valid or not. For a given value of t, the recall is computed as the number of correct recommendations having $C_p \geq t$ divided by 29 (the number of correct recommendations). This is an "approximation" of the *real recall* since we do not know the actual number of correct renamings that are needed in SMOS. In other words, if a correct rename refactoring was not recommended by LEAR, it was not evaluated by the participant and thus is not considered in the computation of the recall.

The precision is computed as the number of correct recommendations having $C_p \geq t$ divided by the number of recommendations having $C_p \geq t$. For example, when considering recommendations having $C_p = 1.0$, we only have three recommended renames, two of which have been accepted by the developer. This results in a recall of 0.07 (2/29) and a precision of 0.67 (2/3)—see Fig. 1.

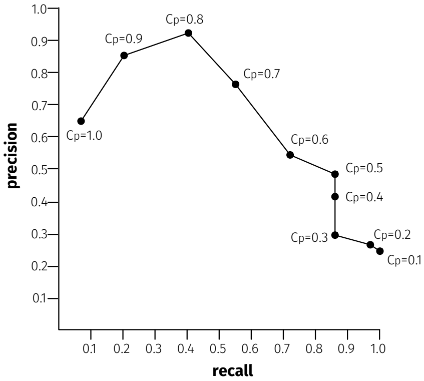

Fig. 1: Precision and recall of the LEAR recommendations when varying C_p

Looking at Fig. 1, we can see that both recall and precision increase moving from $C_p = 1.0$ to $C_p = 0.8$, reaching recall=0.42 (12/29) and precision=0.92 (12/13). This means that only one among the top-13 recommendations ranked by C_p has been considered as not meaningful by the developer. Moving towards lower values of C_p, the recall increases thanks to the additional recommendations considered, while the precision decreases, indicating that the quality of the generated recommendations tend to decrease with lower C_p values (*i.e.,* there are higher chances of receiving a meaningless recommendation for low values of C_p). It is quite clear in Fig. 1 that the likelihood of receiving good rename recommendations when $C_p < 0.5$ is very low.

Based on the results of the performed tuning, we modified our tool in order to generate refactoring recommendations only when $C_c \geq 5$ *and* $C_p \geq 0.5$. This parameter setting will be used for all the projects subject of our evaluation, *i.e.,* no project-specific tuning will be performed. In the evaluation reported in Section IV we will further study the meaningfulness of the generated recommendations of rename refactorings for different values of C_p in the significant range, *i.e.,* varying between 0.5 and 1.0.

IV. Evaluation

This section presents the design and the results of the empirical study we carried out to compare the three previously introduced approaches for rename refactoring.

A. Study Design

The *goal* of the study is to assess the meaningfulness of the rename refactorings recommended by CA-RENAMING, NATURALIZE, and LEAR.

The *perspective* of the study is of researchers who want to investigate the applicability of approaches based on static code analysis (*i.e.,* CA-RENAMING) and on the n-gram language model (*i.e.,* NATURALIZE and LEAR) to recommend rename refactorings. The *context* is represented by *objects*, *i.e.,* five software projects on which we ran the three experimented tools to generate recommendations for rename refactorings, and *subjects*, *i.e.,* seven developers of the *objects* assessing the meaningfulness of the recommended rename refactorings.

To limit the number of refactoring recommendations to be evaluated by the developers, we applied the following "filtering policy" to the experimented techniques:

- LEAR: Given the results of the tuning of the C_p and the C_c indicators, we only consider the recommendations having $C_c \geq 5$ and $C_p \geq 0.50$.
- NATURALIZE: We used the original implementation made available by the authors with the recommended $n = 5$ in the n-gram language model. To limit the number of recommendations, and to apply a similar filter with respect to the one used in LEAR, we excluded all recommendations having a probability lower than 0.5. Moreover, since NATURALIZE is also able to recommend renamings for identifiers used for method names (as opposed to the other two competitive approaches), we removed these recommendations, in order to have a fair comparison.
- CA-RENAMING: No filtering of the recommendations was applied (*i.e.,* all of them were considered). This is due to the fact that, as it will be shown, CA-RENAMING generates a much lower number of recommendations as compared to the other two techniques.

Despite these filters, our study involves a total of 922 manual evaluations of recommendations for rename refactoring. Note also that no comparison will be performed in terms of running time (*i.e.,* the time needed by the techniques to generate the recommendations), since none of them requires more than a few minutes (<5) per system.

1) Research Questions and Context: Our study is steered by the following research question:

- **RQ**$_1$ *Are the rename refactoring recommendations generated by approaches exploiting static analysis and NLP meaningful from a developer's point of view?*

The *object* systems taken into account are five Java systems developed and actively maintained at the University of Molise in the context of research projects or as part of its IT infrastructure. As *subjects*, we involved seven of the developers maintaining these systems. Table II shows size attributes (number of classes and LOCs) of the five systems, the number of developers actively working on them (column "Developers"), the number of developers we were able to involve in our study (column "Participants"), the average experience of the involved participants, and their occupation[3].

As it can be seen we involved a mix of professional developers and Computer Science students at different levels (Bachelor, Master, and PhD). All the participants have at least three years of experience in Java and they are directly involved in the development and maintenance of the object systems.

Therio is a Web application developed and maintained by Master and PhD students. It is currently used for research purposes to collect data from researchers from all around the world. *LifeMipp* is a Web application developed and maintained by a professional developer and a PhD student.

[3] Here "Professional" indicates a developer working in industry.

TABLE II: Context of the study (systems and participants)

System	Type	# of Classes	LOCs	Developers	Participants	Experience (mean)	Occupation
Therio	Web App	79	13K	2	1	7+ years	PhD Student
LifeMipp	Web App	72	7K	2	2	7+ years	Professional; PhD Student
MyUnimolAndroid	Android App	96	27K	4	1	5+ years	Professional
MyUnimolServices	Web Services	100	8K	7	2	3+ years	Bachelor students
Ocelot	Desktop App	182	22K	2	1	7+ years	PhD student

LifeMipp has been developed in the context of an European project and it is currently used by a wide user base. *MyUnimolAndroid* is an Android application developed and maintained by students and professional developers. Such an app is available on the Google PlayStore, it has been downloaded more than 1,000 times, and it is mostly used by students and faculties. *MyUnimolServices* is an open-source software developed and maintained by students and professional developers. Such a system is the back-end of the MyUnimolAndroid app. Finally, *Ocelot* is a Java desktop application developed and maintained by PhD students. At the moment, it is used by researchers in an academic context.

2) Data Collection and Analysis: We run the three experimented approaches (*i.e.,* CA-RENAMING, NATURALIZE, and LEAR) on each of the five systems to recommend rename refactoring operations. Given R the set of refactoring recommended by a given technique on system P, we asked P's developers involved in our study to assess the meaningfulness of each of the recommended refactorings. We did not disclose which tool generated the recommendations to the developers. We adopted the same question/answers template previously presented for the tuning of the LEAR's C_c and C_p indicators. In particular, we asked the developers the question: *Would you apply the proposed refactoring?* with possible answers on a three-point Likert scale: 1 (yes), 2 (maybe), and 3 (no). Again, we clarified the meaning of these three possible answers.

Overall, participants assessed the meaningfulness of 725 rename refactorings, 66 recommended by CA-RENAMING, 357 by NATURALIZE, and 302 by LEAR across the five systems. Considering the number of participants involved (*e.g.,* two participants evaluated independently the recommendations generated for LifeMipp), this accounts for a total of 922 refactoring evaluations, making our study the largest empirical evaluation of rename refactoring tools performed with developers having first-hand experience on the object systems.

To answer our research question we report, for the three experimented techniques, the number of rename refactoring recommendations tagged with *yes*, *maybe* and *no*. We also report the precision of each technique computed in two different variants. In particular, given R the set of refactorings recommended by an experimented technique, we compute:

- $Prec_{yes}$, computed as the number of recommendations in R tagged with a *yes* divided by the total number of recommendations in R. This version of the precision considers as meaningful only the recommendations that the developers would actually implement.
- $Prec_{yes \cup maybe}$, computed as the number of recommendations in R tagged with a *yes* or with a *maybe* divided by the total number of recommendations in R. This version of the precision considers as meaningful also the recommendations indicated by the developers as a valid alternative to the original variable name but not calling for a refactoring operation.

Due to lack of space, we discuss the results aggregated by technique (*i.e.,* by looking at the overall performance across all systems and as assessed by all participants). The tools and raw data are available in our replication package [26].

Finally, we analyze the complementarity of the three techniques by computing, for each pair of techniques (T_i, T_j), the following overlap metrics:

$$correct_{T_i \cap T_j} = \frac{|correct_{T_i} \cap correct_{T_j}|}{|correct_{T_i} \cup correct_{T_j}|}$$

$$correct_{T_i \setminus T_j} = \frac{|correct_{T_i} \setminus correct_{T_j}|}{|correct_{T_i} \cup correct_{T_j}|}$$

$$correct_{T_j \setminus T_i} = \frac{|correct_{T_j} \setminus correct_{T_i}|}{|correct_{T_i} \cup correct_{T_j}|}$$

The formulas above use the following metrics:

- $correct_{T_i}$ represents the set of meaningful refactoring operations recommended by technique T_i;
- $correct_{T_i \cap T_j}$ measures the overlap between the set of meaningful refactorings recommended by T_i and T_j;
- $correct_{T_i \setminus T_j}$ measures the meaningful refactoring operations recommended by T_i only and missed by T_j.

The latter metric provides an indication on how a rename refactoring tool contributes to enrich the set of meaningful refactorings identified by another tool. Such an analysis is particularly interesting for techniques relying on totally different strategies (*e.g.,* static code analysis *vs* NLP) to identify different rename refactoring opportunities. Due to space limitation, we only report the three overlap metrics when considering both the recommendations tagged with *yes* and *maybe* as correct. The overlap metrics obtained when only considering the "*yes* recommendations" as meaningful are available in [26].

B. Results

Table III reports the answers provided by the developers to the question *"Would you apply the proposed rename refactoring?"*. Results are presented by approach, starting with the technique based on static code analysis (*i.e.,* CA-RENAMING [6]) followed by four different variations of NATURALIZE and of LEAR using different thresholds for the confidence of the generated recommendations.

TABLE III: Participants' answers to the question *Would you apply the proposed rename refactoring?*

Approach	Confidence	# recomm.		# yes		# maybe		# no		$Prec_{yes \cup maybe}$	$Prec_{yes}$
		overall	mean	overall	mean	overall	mean	overall	mean		
CA-RENAMING	N/A	80	11.43	21	3.00	30	4.29	29	4.14	63.75%	26.25%
NATURALIZE	>=0.5	459	65.57	76	10.86	99	14.14	284	40.57	38.13%	16.56%
NATURALIZE	>=0.6	319	45.57	59	8.43	67	9.57	193	27.57	39.50%	18.50%
NATURALIZE	>=0.7	185	26.43	35	5.00	43	6.14	107	15.29	42.16%	18.92%
NATURALIZE	>=0.8	88	12.57	20	2.86	21	3.00	47	6.71	46.59%	22.73%
LEAR	>=0.5	380	54.29	111	15.86	140	20.00	129	18.43	66.05%	29.21%
LEAR	>=0.6	296	42.29	99	14.14	112	16.00	85	12.14	71.28%	33.45%
LEAR	>=0.7	186	26.57	67	9.57	69	9.86	50	7.14	73.12%	36.02%
LEAR	>=0.8	130	18.57	55	7.86	50	7.14	25	3.57	80.77%	42.31%

Table III does also report the $Prec_{yes}$ and $Prec_{yes \cup maybe}$ computed as described in Section IV-A2.

General Trends. Before discussing in detail the performance of the experimented techniques, it is worthwhile to comment on some general trend reported in Table III. First of all, *the approaches based on NLP generate more recommendations than* CA-RENAMING. This holds as well when considering the highest confidence threshold we experimented with (*i.e.,* 0.8). Indeed, in this case LEAR generates a total of 130 rename refactorings (on average 18.57 per system) and NATURALIZE 88 (12.57 on average), as compared to the 80 recommended by CA-RENAMING (11.43 on average).

Another consideration is that LEAR *recommends a higher number of refactorings that are accepted by the developers with respect to* NATURALIZE *and to* CA-RENAMING. Overall, 111 rename refactorings recommended by LEAR have been fully accepted with a *yes*, as compared to the 76 by NATURALIZE and 21 by CA-RENAMING.

Also, the higher number of accepted refactorings does not result in a lower precision. Indeed, LEAR does also achieve a higher $Prec_{yes}$ with respect to CA-RENAMING (29.21% *vs* 26.25%) and to NATURALIZE (16.56%). The precision of NATURALIZE is negatively influenced by the extremely high number of recommendations it generates when considering all those having confidence $\geq$ 0.5 (*i.e.,* 459 recommendations).

Finally, LEAR*'s and* NATURALIZE*'s precision is strongly influenced by the chosen confidence threshold*. The values on Table III show an evident impact of the confidence threshold on $Prec_{yes}$ and $Prec_{yes \cup maybe}$ for both the approaches. Indeed, going to the least to the most conservative configuration for the confidence level, $Prec_{yes \cup maybe}$ increases by $\sim$14% (from 66.05% to 80.77%) for LEAR and by $\sim$38% for NATURALIZE (from 38.13% to 76.14%), while $Prec_{yes}$ increases by $\sim$13% for LEAR (from 29.21% to 42.31%) and by $\sim$6% for NATURALIZE (from 16.56% to 22.73%).

These results indicate one important possibility offered by these two approaches based on a similar underlying model: Depending on the time budget developers want to invest, they can decide whether to have a higher or a lower number of recommendations, being informed of the fact that the most restrictive threshold is likely to just generate very few false positives, but also to potentially miss some good suggestions.

Per-project analysis. Table IV reports examples of recommendations generated by the three approaches and tagged with *yes*, *maybe*, and *no*.

TABLE IV: Refactorings tagged with *yes*, *maybe*, and *no*

	System	Original name	Rename	$Conf.$	Tag
CA-RENAMING	LifeMipp	`i`	`insect`	N/A	*yes*
	Therio	`pk`	`idCollection`	N/A	*yes*
	MyUnimolAndroid	`data`	`result`	N/A	*maybe*
	Ocelot	`hash`	`md5final`	N/A	*maybe*
	Ocelot	`navigator`	`this`	N/A	*no*
	MyUnimolAndroid	`fullname`	`fullnameOk`	N/A	*no*
NATURALIZE	Ocelot	`callString`	`macro`	0.92	*yes*
	MyUnimolAndroid	`factory`	`inflater`	0.75	*yes*
	Ocelot	`declaration`	`currentDeclaration`	0.79	*maybe*
	MyUnimolServices	`moduleName`	`name`	0.69	*maybe*
	LifeMipp	`species`	`t`	0.64	*no*
	MyUnimolServices	`username`	`token`	0.91	*no*
LEAR	LifeMipp	`image`	`photo`	1.00	*yes*
	MyUnimolServices	`careerId`	`pCareerId`	0.63	*yes*
	Ocelot	`type`	`realType`	0.91	*maybe*
	LifeMipp	`file`	`fileFullName`	0.67	*maybe*
	Therio	`pUsername`	`pName`	0.59	*no*
	MyUnimolAndroid	`info`	`o`	1.00	*no*

Moving to the assessment performed by participants on each project (data available in our replication package [26]), we found that the accuracy of the recommendations generated by the three tools substantially varies across the subject systems.

For example, on the LifeMipp project, CA-RENAMING is able to achieve very high values of precision, substantially better than the ones achieved by the approaches based on NLP. The refactoring recommendations for the LifeMipp project have been independently evaluated by two developers. Both of them agreed on the meaningfulness of all eight recommendations generated by CA-RENAMING. Indeed, the first developer would accept all of them, while the second tagged five recommendations with *yes* and three with *maybe*. NATURALIZE and LEAR, instead, while able to recommend a higher number of *yes* and *maybe* recommendations as opposed to CA-RENAMING (on average 19 for NATURALIZE and 22 for LEAR *vs* the 8 for CA-RENAMING), present a high price to pay in terms of false positives to discard (0 false positives for CA-RENAMING as compared to 49 for NATURALIZE and 19 for LEAR). Such a cost is strongly mitigated when increasing the confidence threshold. Indeed, when only considering recommendations having confidence $\geq$ 0.8, the number of false positives drops to 1 (first developer) or 0 (second developer) for LEAR and to 8 or 6 for NATURALIZE.

However, LEAR and NATURALIZE still keep an advantage in terms of number of *yes* and *maybe* generated recommendations (13 and 14—depending on the developer—for LEAR, and 12, for both developers, for NATURALIZE). A similar trend has also been observed for MyUnimolServices.

When run on MyUnimolAndroid, CA-RENAMING only recommends three rename refactorings, two tagged with a *maybe* and one discarded (*no*). NATURALIZE generates 65 recommendations, with nine *yes*, 14 *maybe*, and 42 *no*. Finally, LEAR generates 35 suggestions, with six *yes*, 12 *maybe*, and 17 *no*.

This is the only system in which we did not observe a clear trend between the quality of the refactoring recommended by LEAR and the value used for the C_p threshold. Indeed, the precision of our approach is not increasing with the increase of the C_p value. This is due to the fact that the developer involved in the evaluation of the refactoring for the MyUnimolAndroid rejected with a *no* seven recommendations having $C_p \geq 0.8$.

We asked the developer for further comments to check what went "wrong" for this specific system, and in particular we asked to comment on each of these seven cases. Some of the explanations seemed to indicate more a *maybe* recommendation rather than the assigned *no*. For example, our approach recommended with $C_p = 0.9$ and $C_c = 54$ the renaming *activity* → *navigationDrawer*. The developer explained that the *activity* identifier refers to an object of `FragmentActivity` that is casted as a `NavigationDrawer` and, for this reason, he prefers to keep the *activity* name rather than the recommended one. Another false positive indicated by the developer was renaming *info* → *o*, where *info* is a method parameter of type `Object`. LEAR learned from the MyUnimolAndroid's trigrams that the developers tend to name a parameter of type `Object` with *o*. This is especially true in the implementation of `equals` methods. Thus, while the renaming would have been consistent with what is present in the system, the developer preferred to keep the original name as being "*more descriptive*", rejecting the recommendation. MyUnimolAndroid is also the only system in which NATURALIZE achieves a higher precision than LEAR when considering the most restrictive confidence (*i.e.,* ≥ 0.8).

Finally, on the Therio and on the Ocelot projects, LEAR substantially outperforms the two competitive approaches. On Therio, CA-RENAMING achieves $Prec_{yes} = 0.33$ and $Prec_{yes \cup maybe} = 0.47$, as compared to the $Prec_{yes} = 0.37$ and $Prec_{yes \cup maybe} = 0.74$ achieved by LEAR when considering only recommendations having $C_p \geq 0.6$. LEAR also generates a much higher number of *yes* (35 *vs* 5) and *maybe* (13 *vs* 2) recommendations. Examples of recommendations generated by LEAR and accepted by the developers include *pk* → *idTaxon* and *o* → *occurrences*, while an example of rejected recommendation is *pUsername* → *pName*. NATURALIZE also achieves its best performance on Therio when considering all recommendations having confidence ≥ 0.6 ($Prec_{yes} = 0.35$ and $Prec_{yes \cup maybe} = 0.74$), but with a lower number of *yes* (23) and *maybe* (8) recommendations with respect to LEAR. A similar trend is also observed on Ocelot, where LEAR is able to recommend 89 renamings with a $Prec_{yes \cup maybe} = 0.93$.

Overlap Metrics Analysis. Table V reports the three overlap metrics between the experimented techniques.

TABLE V: Overlap metrics

T_i	T_j	$correct_{T_i \cap T_j}$	$correct_{T_i \setminus T_j}$	$correct_{T_j \setminus T_i}$
CA-RENAMING	LEAR	1.00%	16.05%	82.94%
CA-RENAMING	NATURALIZE	0.00%	22.57%	77.43%
LEAR	NATURALIZE	4.16%	57.21%	38.63%

The overlap in terms of meaningful recommendations provided by the different tools is extremely low; 1% between CA-RENAMING and LEAR, 0% between CA-RENAMING and NATURALIZE, and 4% between LEAR and NATURALIZE. While the low overlap between the techniques using static code analysis and NLP is somehow expected, the 4% overlap observed between LEAR and NATURALIZE is surprising considering the fact that LEAR is inspired by the core idea behind NATURALIZE. This means that the differences between the two techniques described in Section III (*e.g.,* only considering the lexical tokens in the language model as opposed to using all tokens) have a strong impact on the generated recommendations. While this was already clear by the different performance provided by the two approaches (see Table III), it is even more evident from Table V.

LEAR is able to recommend 82.94% of meaningful renamings that are not identified by CA-RENAMING, and 57.21% that are not recommended by NATURALIZE. However, there is also a high percentage of meaningful rename refactorings recommended by CA-RENAMING (16.05%) and NATURALIZE (38.63%) but not identified by LEAR. This confirms the very high complementarity of the different techniques, paving the way to novel rename refactoring approaches based on their combination, which will be investigated in our future work.

V. Threats to Validity

Threats to *construct validity* are mainly related to how we assessed the developers' perception of the refactoring meaningfulness. We asked developers to express on a three-point Likert scale the meaningfulness of each recommended refactoring making sure to carefully explain the meaning of each possible answer from a practical point of view.

Threats to *internal validity* are represented, first of all, by the calibration of the LEAR confidence C_p and C_c indicators. We performed the calibration of these indicators on one project (SMOS) not used in the LEAR's evaluation, by computing the recall *vs* precision curve for different possible values of the C_p indicator. This was not really needed for the C_c indicator, for which we just observed the unreliability of the recommendations having $C_c < 5$. Concerning the other approaches, for the NATURALIZE's n-gram model parameter we adopted the one used by its authors (*i.e.,* $n = 5$) and we relied on their implementation of the approach. To limit the number of refactoring recommendations, we excluded the ones having a probability lower than 0.5. This choice certainly does not penalize NATURALIZE, since we are only considering the best recommendations it generates. As for CA-RENAMING, we used our own implementation (available in [26]).

Threats to *external validity* are related to the set of chosen objects and to the pool of participants. Concerning the objects, we are aware that our study is based on refactorings recommended on five Java systems only and that the considered systems, while not trivial, are generally of small-medium size (between 7 and 27 KLOC). Also, we were only able to involve in our study seven developers. Still, as previously said, (i) we preferred to limit our study to developers having a first-hand experience with the object systems, rather than inviting also external developers to take part in our study, and (ii) despite the limited number of systems and developers, our results are still based on a total of 922 manual inspections performed to assess the quality of the refactorings.

VI. Conclusion

We assessed the meaningfulness of recommendations generated by three approaches—two existing in the literature (*i.e.,* CA-RENAMING [6] and NATURALIZE [7]) and one presented in this paper (*i.e.,* LEAR)—promoting a consistent use of identifiers in code. The results of our study highlight that:

1. Overall, LEAR achieves a higher precision, and it is able to recommend a higher number of meaningful refactoring operations with respect to the competitive techniques.

2. While being the best performing approach, LEAR still generates a high number of false positives, especially when just considering as meaningful the recommendations tagged with a *yes* by the developers (*i.e.,* the ones they would actually implement). This means that there is large room for improvement in state-of-the-art tools for rename refactoring.

3. The experimented approaches have unstable performance across the different systems. Indeed, even if LEAR is, overall, the approach providing the most accurate recommendations, it is not the clear winner on all the object systems. This indicates that there are peculiarities of the software systems that can influence the performance of the three techniques.

The above observations will drive our research agenda, including: (i) revising our approach to exploit more information (*e.g.,* data flow graph) to increase its performance, and (ii) studying the characteristics of the software systems that influence the accuracy of the rename refactoring tools.

References

[1] F. Deissenboeck and M. Pizka, “Concise and consistent naming,” *Software Quality Journal*, vol. 14, no. 3, p. 261, 2006.

[2] ——, “Concise and consistent naming,” *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

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

[4] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc, “Can lexicon bad smells improve fault prediction?” in *Proceedings of WCRE 2012 (19th Working Conference on Reverse Engineering)*. IEEE, 2012, pp. 235–244.

[5] V. Arnaoudova, L. Eshkevari, R. Oliveto, Y.-G. Gueheneuc, and G. Antoniol, “Physical and conceptual identifier dispersion: Measures and relation to fault proneness,” in *Proceedings of ICSM 2010 (26th International Conference on Software Maintenance)*. IEEE, 2010, pp. 1–5.

[6] A. Thies and C. Roth, “Recommending rename refactorings,” in *Proceedings of RSSE 2010 (2nd International Workshop on Recommendation Systems for Software Engineering)*. ACM, 2010, pp. 1–5.

[7] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 281–293.

[8] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “Whats in a name? a study of identifiers,” in *Proceedings of ICPC 2006 (14th International Conference on Program Comprehension)*. IEEE, 2006, pp. 3–12.

[9] D. Lawrie, H. Feild, and D. Binkley, “Quantifying identifier quality: an analysis of trends,” *Empirical Software Engineering*, vol. 12, no. 4, pp. 359–388, 2007.

[10] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, “Exploring the influence of identifier names on code quality: An empirical study,” in *Proceedings of CSMR 2010 (14th European Conference on Software Maintenance and Reengineering)*. IEEE, 2010, pp. 156–165.

[11] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 5–18, 2012.

[12] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, “Tidier: an identifier splitting approach using speech recognition techniques,” *Journal of Software: Evolution and Process*, vol. 25, no. 6, pp. 575–599, 2013.

[13] A. Corazza, S. Di Martino, and V. Maggio, “Linsen: An efficient approach to split identifiers and expand abbreviations,” in *Proceedings of ICSM 2012 (28th International Conference on Software Maintenance)*. IEEE, 2012, pp. 233–242.

[14] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, “Mining source code to automatically split identifiers for software analysis,” in *Proceedings of MSR 2009 (6th IEEE International Working Conference on Mining Software Repositories)*. IEEE, 2009, pp. 71–80.

[15] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker, “Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools,” in *Proceedings of MSR 2008 (5th IEEE International Working Conference on Mining Software Repositories)*. ACM, 2008, pp. 79–88.

[16] D. Lawrie and D. Binkley, “Expanding identifiers to normalize source code vocabulary,” in *Proceedings of ICSM 2011 (27th International Conference on Software Maintenance)*. IEEE, 2011, pp. 113–122.

[17] S. P. Reiss, “Automatic code stylizing,” in *Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering*. Atlanta, Georgia, USA: ACM Press, 2007, pp. 74–83.

[18] F. Corbo, C. Del Grosso, and M. Di Penta, “Smart Formatter: Learning coding style from existing source code,” in *Proceedings of 23rd IEEE International Conference on Software Maintenance*. Paris, France: IEEE CS Press, Oct. 2007, pp. 525–526.

[19] B. Caprile and P. Tonella, “Restructuring program identifier names,” in *Proceedings of ICSM 2000 (2000 International Conference on Software Maintenance)*, 2000, pp. 97–107.

[20] E. W. Høst and B. M. Østvold, “Debugging method names,” in *Proceedings of ECOOP 2009 (23rd European Conference on Object-Oriented Programming)*. Springer, 2009, pp. 294–317.

[21] A. Feldthaus and A. Møller, “Semi-automatic rename refactoring for javascript,” in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 323–338.

[22] P. Jablonski and D. Hou, “Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide,” in *Proceedings of ETX 2007 (2007 OOPSLA workshop on eclipse technology eXchange)*. ACM, 2007, pp. 16–20.

[23] M. L. Collard, M. J. Decker, and J. I. Maletic, “srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration,” in *Proceedings of ICSM 2013 (29th International Conference on Software Maintenance)*. IEEE, 2013, pp. 516–519.

[24] B. Lin, L. Ponzanelli, A. Mocci, G. Bavota, and M. Lanza, “On the uniqueness of code redundancies,” in *Proceedings of ICPC 2017 (25th International Conference on Program Comprehension)*. IEEE, 2017.

[25] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software,” in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[26] Authors, “Replication package.” https://scam-identifier.github.io/replication.zip.