2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

# Nomen est Omen: Exploring and Exploiting Similarities between Argument and Parameter Names

Hui Liu[1], Qiurong Liu[1], Cristian-Alexandru Staicu[2], Michael Pradel[2], Yue Luo[1]
[1]School of Computer Science and Technology, Beijing Institute of Technology, China
[2]Department of Computer Science, TU Darmstadt, Germany
{liuhui08,liuqiurong}@bit.edu.cn, cris.staicu@gmail.com, michael@binaervarianz.de, 102286165@qq.com

## ABSTRACT

Programmer-provided identifier names convey information about the semantics of a program. This information can complement traditional program analyses in various software engineering tasks, such as bug finding, code completion, and documentation. Even though identifier names appear to be a rich source of information, little is known about their properties and their potential usefulness. This paper presents an empirical study of the lexical similarity between arguments and parameters of methods, which is one prominent situation where names can provide otherwise missing information. The study involves 60 real-world Java programs. We find that, for most arguments, the similarity is either very high or very low, and that short and generic names often cause low similarities. Furthermore, we show that inferring a set of low-similarity parameter names from one set of programs allows for pruning such names in another set of programs. Finally, the study shows that many arguments are more similar to the corresponding parameter than any alternative argument available in the call site's scope. As applications of our findings, we present an anomaly detection technique that identifies 144 renaming opportunities and incorrect arguments in 14 programs, and a code recommendation system that suggests correct arguments with a precision of 83%.

## CCS Concepts

**•Software and its engineering → Software development techniques; Maintaining software;**

## Keywords

Empirical study, name-based program analysis, identifier names, static analysis, method arguments

## 1. INTRODUCTION

Identifier names chosen by developers convey information about the semantics of a program [21], but many program analyses ignore identifier names. As a result, running an analysis on a human-written program with meaningful identifier names and on an equivalent program where all identifiers are consistently replaced with arbitrary strings gives exactly the same result. However, ignoring names discards a valuable source of information that may provide hints that are otherwise unavailable to an analysis. To exploit this information, recent work uses identifier names to infer API specifications [33, 25], to identify mismatches between a method name and the method's implementation [16], to synthesize code completions [29], to predict syntactic and semantic properties of programs [28], to suggest identifier names [2, 3], and to detect incorrectly ordered method arguments of the same type [26, 27].

Despite these recent approaches, little is known about the properties of identifier names in real-world software and about how one could exploit these properties. Are identifier names that refer to semantically related values similar? Is it possible to predict from identifier names which variable, field, or method a developer will use next? How prevalent are names that convey little or no semantic information, such as generated variable names or very short names, and is there a way to identify them? Addressing these questions is valuable because it may pave the road for name-based analyses that complement traditional program analysis. For example, one could exploit the similarity of names to complete pieces of code automatically, to warn developers about anomalies that may correspond to code worth changing, or to infer documentation from names.

This paper focuses on the lexical similarity of arguments and parameters of methods, which is one prominent situation where identifier names may provide otherwise missing semantic links. We say *argument* to values passed to a method at a call site, and we say *parameter* to the formal parameter in the method's definition. Since an argument and its corresponding parameter often refer to the same instance, we hypothesize that their names are often similar.

### 1.1 Research Questions

To evaluate this hypothesis, we conduct an empirical study that addresses the following research questions:

*RQ1*: How similar are argument names to the names of their corresponding parameters? Answering this question will help to decide whether exploring similarities between argument names and parameter names is worthy.

*RQ2*: How long are argument names and parameter names in real-world Java programs, and how does the length relate to their similarity? Answering this question may help es-



*ICSE '16, May 14 - 22, 2016, Austin, TX, USA*



ISBN 978-1-4503-3900-1/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884841

timate how much confidence one can have into similarities between names of particular lengths.

*RQ3*: How often does an overriding method change the parameter names compared to the names in the overridden method? Answering this question helps estimate how often comparing arguments to the parameter names of the statically resolved call target is sufficient, and how often considering the dynamic call target would yield different results.

*RQ4*: Why are some arguments dissimilar to corresponding parameters? Answering this question my help applications that exploit name similarities to ignore particular kinds of dissimilarities.

*RQ5*: If parameters with specific names, such as `arg0`, are frequently assigned with dissimilar arguments in sample applications, are parameters with the same name frequently assigned with dissimilar arguments in other applications? If yes, can we build a set of *low-similarity parameters* that are likely to be assigned with dissimilar arguments? Answering this question may help approaches that exploit the similarity of names, such as name-based code completion or anomaly detection, to reduce false positives by ignoring arguments assigned to low-similarity parameters.

*RQ6*: How often is the argument chosen by the developer more similar to the corresponding parameter than any of its potential alternatives? Answering this question will help to estimate the accuracy of approaches that exploit the similarity of names, such as code completion.

## 1.2 Summary of Findings

To address these questions, we empirically study 609,489 named arguments in 60 popular open-source Java programs. The main findings of the study are the following. For RQ1, we find that the distribution of the lexical similarity is a U-shape: the similarity is either extremely high or extremely low. Many arguments (31%) exactly match their corresponding parameter. In contrast, the majority of those arguments that are not similar to their corresponding parameters are very dissimilar (51% have a similarity of 0%).

For RQ2, we find that 84% of all argument names and 70% of all parameter names have at least four characters. Almost all argument names (91%) and most parameter names (81%) are composed of at most three terms. On average over all studied parameters, the similarity to their arguments increases with the length of parameter names and with the number of terms used in the parameter name. However, there is no strong correlation between the length of names and the similarity between individual pairs of arguments and parameters. These results suggest that one cannot infer the similarity between an argument and a parameter from the length of their names.

For RQ3, we find that most of all overriding methods (92%) have exactly the same list of parameter names as their overridden method. That is, comparing arguments to the parameters of a statically computed call target is sufficient in most cases, even though a call may be dispatched to a different target method at runtime.

For RQ4, we find that the main reason for dissimilar arguments and parameters are short parameter names. Among the 310,814 named arguments whose similarity with their corresponding parameter is zero, 23% are assigned to parameters named with a single character, and 42% of them are assigned to parameters named with no more than three characters. Another reason for dissimilar pairs of arguments and parameters are generic data collection operations. Among the 310,814 named arguments whose lexical similarity with their corresponding parameters is zero, 14% are assigned to parameters named `index`, `item`, `key`, or `value`. Such names are common in methods manipulating data collections.

For RQ5, we find that most dissimilar pairs of argument names and parameters (75%) are due to a small set of parameters that occur again and again across programs, such as `arg0`. That is, extracting a set of parameter names to ignore from sample programs helps finding parameters that are likely to be associated with dissimilar arguments in other programs.

Finally, for RQ6, we find that most argument names (55%) that are not associated with low-similarity parameters are more similar to their corresponding parameter name than any other argument that a programmer could use in the current scope. The figure is even up to 78% for those arguments whose lexical similarity with their corresponding parameters is at least 0.67. That is, analyzing the similarity of argument names and parameter names can help in deciding which argument to use and in detecting incorrect or otherwise suspicious arguments.

## 1.3 Applications

Our results suggest several research directions for exploiting identifier names to support and further automate software development tasks, and we explore two such directions. First, we present a static anomaly detection technique that warns about argument names that seem not to match their corresponding parameter name. The basic idea is to report a warning when the developer could use another argument than the current one, and when this change would make the argument name and parameter name significantly more similar to each other. We apply the analysis to 10 programs, where it finds 6 known incorrect arguments, 3 previously unknown incorrect arguments, and 127 renaming opportunities with a precision of 80%. Second, we present a name-based recommendation system that suggests an argument while a developer writes code that calls a method. The basic idea is to recommend an argument from a set of potential arguments so that the recommended argument is the most similar to the corresponding parameter. The approach recommends correct arguments with a precision of 83%.

In summary, this paper contributes the following:

- The first extensive empirical study of argument names and parameter names in real-world Java programs.
- Empirical evidence showing that most argument names and parameter names are either similar to each other or can be easily filtered, and that names help in deciding which argument to assign to a parameter.
- Two practical applications of our findings, anomaly detection and argument recommendation, along with experimental results that show that the applications are effective.

# 2. SETUP OF THE STUDY

## 2.1 Methodology

To study the lexical similarity of identifier names involved in method calls, we compare named method arguments and

method parameters with a string similarity metric. The first step of the study is to extract identifier names from the source code of the subject applications. To this end, an AST-based, static analysis extracts from each formal parameter in a method definition the identifier of the parameter, called *parameter name*. Furthermore, the analysis extracts at each call site the names of particular kinds of arguments, called *argument names*. Specifically, the analysis considers the following expressions that may be passed as arguments:

- For a variable, the argument name is simply the variable name.
- For a call expression, the argument name is the name of the called method. That is, we ignore the receiver object and any arguments of the call. For example, if the return value of a method call $student.firstName()$ is passed as an argument, then the argument name is $firstName$.
- For a field access expression, the argument name is the name of the field, again ignoring the receiver object. For example, if the argument is *student.id*, then the argument name is *id*.
- For the *this* keyword, the argument name is the name of the class of which *this* is an instance. For example, if in a method invocation *print(this)* the argument *this* refers to an instance of class *Student* then the argument name is *Student*.

All other arguments, such as complex expressions, are ignored in the study.

To compare argument names with parameter names, the analysis matches each call site with a method definition. This matching is based on the statically known target method to which the call resolves. As a result, each parameter name is associated with a set of argument names.

Based on the extracted argument and parameter names, we compute the similarity of two names as follows. To measure the lexical similarity between names, we decompose each identifier name into a sequence of terms, noted as $terms(arg)$. The decomposition is based on underscores and capital letters, assuming that the name follows the popular camel case or snake case naming convention. We measure the *lexical similarity* between argument $arg$ and parameter $par$ as follows:

$$lexSim(arg, par) = \frac{|comterms(arg, par)| + |comterms(par, arg)|}{|terms(arg)| + |terms(par)|} \quad (1)$$

$comterms(n_1, n_2)$ is the longest subsequence of $terms(n_1)$, where each term in the subsequence appears in $terms(n_2)$. For example, $lexSim(\text{“}length\text{”}, \text{“}inputLength\text{”}) = \frac{1+1}{1+2} = 67\%$ and $lexSim(\text{“}fieldLength\text{”}, \text{“}fieldLength\text{”}) = \frac{2+2}{2+2} = 100\%$. The measure may assign a similarity of 1 to non-equal names, such as “$fooBar$” and “$barFoo$”, which may appear unintuitive but is not a problem in practice.

## 2.2 Subject Applications

We search for the most popular open-source Java applications from SourceForge, and select the top 60 resulting applications for investigation. Sizes of such applications vary from 2,893 to 570,384 non-blank lines of source code. In total, the subject applications are composed of 5,841,635 lines of code.

From these subject applications, we extract all named arguments. In total, we get 609,489 named arguments from these applications. For each such argument, we compute the lexical similarity between it and its corresponding parameter according to Formula 1.

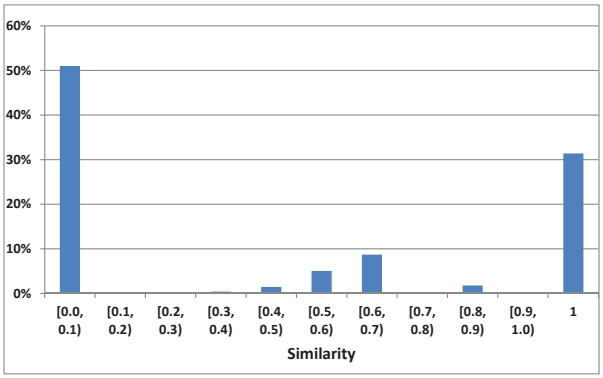


**Figure 1: Distribution of lexical similarity between arguments and parameters**

# 3. RESULTS OF THE STUDY

## 3.1 RQ1: Distribution of Lexical Similarity

To address the question how similar argument and parameter names are, we compute the similarity between all argument names and the names of their corresponding parameters. Figure 1 shows the distribution of the similarity. The figure shows that the distribution has a U-shape: similarity is either very high or very low. In 31% of all cases, the similarity is equal to 1, whereas in 51% of all cases, the similarity is 0 (i.e., the names share no common terms).

From the figure, we also observe that the number of arguments whose similarity with corresponding parameters belongs to [0.1,0.2), [0.2,0.3), [0.3,0.4), [0.7,0.8), and [0.9,1.0) is extremely small. One of the reasons is that the similarity is computed based on terms, and thus the similarity is discrete. If two names contain no common terms, the similarity is zero. Otherwise, the numerator of Formula 1 is at least 2. That is, to obtain a similarity smaller than 0.2, the denominator must be greater than $2/0.2 = 10$. However, it is not so common that the length of two identifier names is longer than 10 terms. Consequently, the number of arguments whose similarity with corresponding parameters belongs to [0.1,0.2) is extremely small. The same is true for other intervals, e.g., [0.2,0.3), [0.3,0.4), [0.7,0.8), and [0.9,1.0).

We conclude from these results that the similarity of argument and parameter names is worth exploring further, because a significant part of all arguments is very similar to its corresponding parameter.

## 3.2 RQ2: Length of Names

To address the question how long argument names and parameter names in real-world Java programs are, we measure for each name the number of characters and the number of terms in the name. Figure 2 shows the results. The

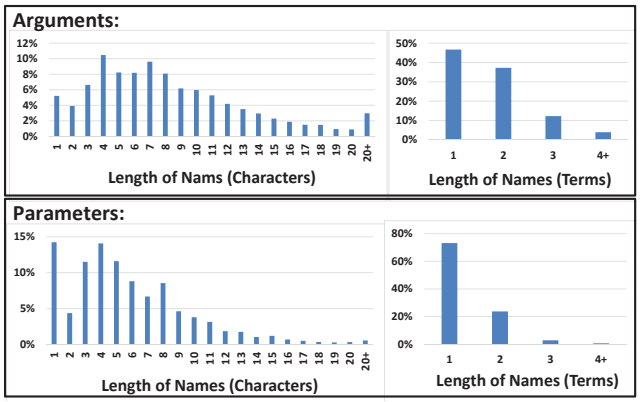


**Figure 2: Length of argument names and parameter names.**

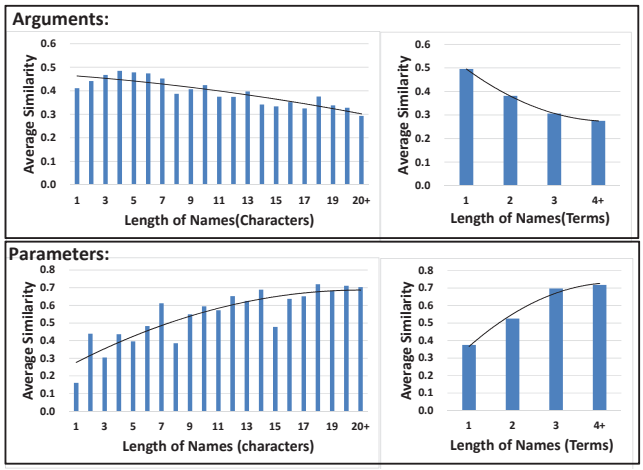


**Figure 3: Correlation between length of names and average similarity (with polynomial trendlines)**

left-hand side of the figure shows that most names are composed of no more than 10 characters. For example, 63% of the argument names contain 3 to 10 characters. 80% of the parameter names contains no more than 8 characters. The right-hand side of the figure shows that most names are composed of a small number of terms. 99% of the parameter names and 96% of the argument names are composed of at most three terms. Single-term names account for 45% of argument names and 73% of parameter names.

The average length of argument names (1.7 terms) is higher than that of parameter names (1.3 terms). One of the reasons why argument names contain more terms are method invocations whose return value is passed as an argument. For a single-term parameter, e.g., `rectangle`, an argument could be a method invocation, e.g., `createRectangle()`.

We also investigate whether the length of parameter names or argument names influences the similarity between arguments and parameters. The results are presented in Figure 3. The figure shows that the average similarity increases when the length of parameter names increase. In contrast, the similarity decreases while the length of argument names increases. Even though, on average over all studied arguments and parameters, the length of names influences the similarity, the length of parameter names and argument names is only weakly correlated to similarity for individual pairs of arguments and parameters. The correlation coefficient is 0.15 (parameter names) and -0.18 (argument names), respectively. One of the reasons for this seemingly contradictory result is that even for the argument (or parameter) names of the same length, the similarity between arguments and parameters may vary dramatically.

We conclude from these results that the length of parameter names does influence the similarity between their names. This finding suggests that developers should aim for expressive parameter names with multiple terms (if appropriate).

## 3.3 RQ3: Parameter Names of Overridden Methods

For this study, we resolve invoked methods based on the statically known type of the receiver object. However, static analysis may resolve method invocations incorrectly because the dynamic receiver type may be a subtype of the static receiver type that overrides the called method. To investigate to what extent such mis-resolution may influence the measured similarity between arguments and parameters, we measure how often overriding methods use different parameter names than the overridden method.

In total over all applications, we find that most of the overriding methods (92%) have exactly the same parameter list (including the same parameter names) as the methods they override. In other words, in most cases the fact that static method resolution may be incorrect does not influence the measured similarity between arguments and parameters.

## 3.4 RQ4: Reasons for Dissimilarity

To address the question why some arguments are dissimilar to their corresponding parameter, we further analyze all pairs of argument name and parameter name that have a similarity of 0. In total, there are 310,814 such pairs. We randomly sample 200 of them and manually inspect them. Moreover, we validate the results of the manual inspection on the whole population of 310,814 pairs of names.

The manual analysis leads to two findings: First, very short parameter names are the major reason for the dissimilarity between arguments an parameters. 21% of the arguments are assigned to parameters named with a single character, e.g., `s` and `i`, and 40.5% of the arguments are assigned to parameters named with at most three characters. Such short parameter names often convey little semantics and therefore cause dissimilarities with the corresponding argument names.

Second, generic parameter names of methods of collection classes are another reason for the dissimilarity between arguments and parameters. 14% of the arguments are assigned to parameters named `index`, `item`, `key`, or `value`. Such parameter names are popular in methods that manipulate data collections. Although such parameter names are meaningful, their corresponding arguments are usually dissimilar to them because their arguments are concrete value or indexes. For example, an invocation of method `List.add(int index, Object value)` may be `add(i, newElement)`.

To investigate whether the analysis results on the 200 sample arguments can be generalized, we validate the results on the entire population of argument names that have no similarity with the corresponding parameter name. The validation results are as follows:

- 71,499 (23%) of the 310,814 named arguments were assigned to parameters named with a single character. 130,708 (42%) out of the 310,814 named arguments were assigned to parameters named with no more than 3 characters. These data are similar to the analysis results on the sample.
- 36,886 (12%) of the 310,814 named arguments were assigned to `index`, `item`, `key`, or `value`, which is similar to the analysis results on the sample.

We conclude from the results that short parameter names and generic names are main reasons for dissimilarities between argument names and parameter names. This finding can benefit applications of name similarities, which may, e.g., ignore such parameters.

## 3.5 RQ5: Filtering Parameters with Low Similarity

The following addresses the question whether one can build a set of low-similarity parameters from a corpus of sample applications. We present a technique for computing such a set and assess the effect of filtering pairs of argument names and low-similarity parameter names.

Given a corpus of sample applications, our approach for identifying low-similarity parameters has three steps. First, we cluster all argument names in the sample applications by their corresponding parameter names. If the parameters associated with two arguments have the same name, then we assign both arguments into the same cluster. That is, each cluster is associated with a unique parameter name. Second, for each cluster, we calculate the average similarity $s$ between arguments in this cluster and their corresponding parameters. Finally, if the average similarity $s$ of a cluster is smaller than 0.5, we add the parameter name associated with this cluster to the set of low-similarity parameters. For example, for two calls `m(a,x)` and `m(a,y)` of the method `m(A a, B b)`, we extract two clusters $C_a = [a, a]$ and $C_b = [x, y]$. The average similarities for $C_a$ and $C_b$ are $avg(lexSim(\text{“}a\text{”}, \text{“}a\text{”}), lexSim(\text{“}a\text{”}, \text{“}a\text{”})) = 1$ and $avg(lexSim(\text{“}x\text{”}, \text{“}b\text{”}), lexSim(\text{“}y\text{”}, \text{“}b\text{”})) = 0$, respectively. Hence, we add `b` to the set of low-similarity parameters.

To assess to what extent extracting low-similarity parameters from sample programs can help find parameters that are likely to be associated with dissimilar arguments in other programs, we carry out a k-fold cross-validation on the 60 subject applications (k=6). The applications are randomly partitioned into 6 groups, notated as $Sag_i$ $(i = 1 \ldots 6)$, where each group is composed of 10 subject applications. For the $i$th cross-validation, we consider all subject applications except for those in $Sag_i$ as the corpus of training applications, and we consider the applications in $Sag_1$ as the validation applications.

After identifying low-similarity parameters from a corpus of sample applications, we compute all argument names in the remaining applications whose corresponding parameters are low-similarity parameters. We call such arguments *filtered out arguments*. Figure 4 shows the distribution of the similarity between filtered out arguments and their corresponding parameters. The figure shows that for most (73% on average) of the filtered out arguments the similarity between them and their corresponding parameters is zero. Note that this percentage is significantly higher than when considering all arguments (Figure 1). The observation holds in all of the six rounds of cross-validations regardless of the changes on both training data and validation data.

**Table 1: Influence of ignoring arguments associated with low-similarity parameters.**

| Similarity | Arguments ($n_1$) | Filtered out arguments ($n_2$) | $n_2/n_1$ |
|---|---|---|---|
| $[0.0, 0.1)$ | 310,814 | 233,311 | 75% |
| $[0.1, 0.2)$ | 4 | 0 | 0% |
| $[0.2, 0.3)$ | 1,059 | 300 | 28% |
| $[0.3, 0.4)$ | 2,481 | 527 | 21% |
| $[0.4, 0.5)$ | 8,799 | 1,919 | 22% |
| $[0.5, 0.6)$ | 30,707 | 9,821 | 32% |
| $[0.6, 0.7)$ | 53,023 | 15,947 | 30% |
| $[0.7, 0.8)$ | 344 | 4 | 1% |
| $[0.8, 0.9)$ | 10,827 | 730 | 7% |
| $[0.9, 1.0)$ | 55 | 0 | 0% |
| 1 | 191,376 | 56,849 | 30% |
| Total | 609,489 | 319,408 | 52% |

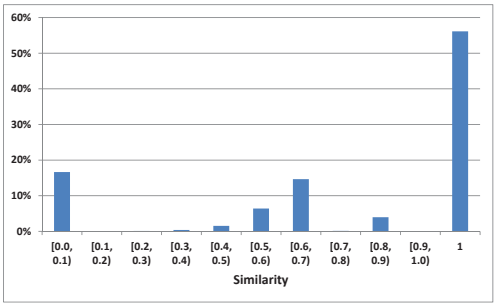


**Figure 5: Distribution of similarity parameters and arguments that are not associated with low-similarity parameters.**

The above results show that low-similarity parameters are an effective means for filtering arguments across applications. In the remainder if the paper, we use low-similarity parameters computed from all subject applications. Table 1 illustrates how filtering out arguments associated with low-similarity parameters changes the distribution of similarity between arguments and parameters. From the table, we observe that arguments that are less similar to their parameters are more likely to be filtered out. For example, the filtering removes 75% of all arguments whose similarity with their parameters is zero, whereas the filtering removes only 29% of all arguments whose similarity with parameters is not zero.

Figure 5 shows the distribution of similarity between parameters and arguments that are not filtered out arguments. By comparing Figure 5 with Figure 1, we observe that the distribution of lexical similarity between arguments and parameters has been reshaped dramatically by filtering based on low-similarity parameters: The ratio of arguments that are dissimilar to their parameters decreases and the ratio of arguments that are similar to their parameters increases.

We conclude from the results that computing low-similarity parameters from a corpus of sample applications is an effective means to predict whether a parameter will be dissimilar to its arguments. This finding enables approaches that exploit name similarities to improve their precision by ignoring low-similarity parameters.

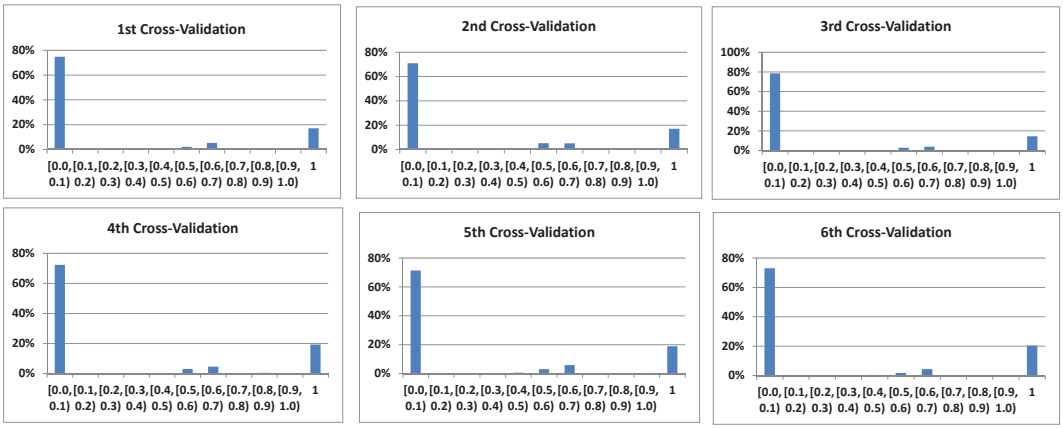


**Figure 4: Distribution of similarity between parameters and arguments associated with low-similarity parameters.**

## 3.6 RQ6: Picking Among Alternative Arguments

To address the question how similar the argument chosen by the developer is to other available arguments, we compare each argument against its potential alternatives. To this end, we define which arguments could be used by a developer and then compute which of these possible arguments is the most similar to the parameter.

*Definition 1. Potential alternatives* of argument *arg* are:

- If *arg* is a local variable or a field of the enclosing class, all fields of the enclosing class and local variables that are available at the location are in the set of potential alternatives.
- If *arg* is a field access or a method invocation without arguments, then all field accesses and method invocations without arguments that have the same receiver object are potential alternatives.
- If *arg* is a method invocation with arguments, method invocations with the same receiver object and the same arguments are in the set of potential alternatives.

For example, for an argument `a.b.foo`, the following are potential alternatives: `a.b.bar` and `a.b.getBar()`. Likewise, for a method invocation `x.getSize(y)`, the method invocation `x.length(y)` is a potential alternative.

Replacing an argument with a potential alternative may introduce syntactical errors or type errors. For example, some alternatives may not be available in the current scope (e.g., private fields), or their types may be incompatible with the parameter's type. We exclude such invalid potential alternatives from the comparison, and define *alternative arguments* as follows:

*Definition 2.* An *alternative argument* of argument *arg* is a potential alternative that does not introduce new syntactical or type errors while replacing *arg*.

Among the alternative arguments *alt_args*, we call arguments that have the greatest lexical similarity with the corresponding parameter *par* a *most similar argument*:

*Definition 3.* An alternative argument $m\_alt \in alt\_args$ is a *most similar argument* if for any alternative argument $any\_alt \in alt\_args$, the following inequation holds:
$lexSim(m\_alt, par) \geq lexSim(any\_alt, par)$

To investigate how often the argument chosen by the developer is more similar to the corresponding parameter than any of its potential alternatives, we compare each argument name in the applications to the names of its alternative arguments (especially the most similar argument).

50% of the 609,489 studied arguments do not have any alternative arguments, i.e., they are the only argument that is available in the scope of the method call and that is type-compatible with the parameter. Among the 304,387 arguments with alternatives, only a small number (13.9%= 42,339/304,387) have an alternative that is more similar to the parameter than the current argument. In other words, 86.1% of all arguments are among the most similar of all possible arguments. For 27%=82,158/304,387 of the arguments with an alternative, the current argument is even strictly more similar to the parameter than any of its alternatives. This number increases to 78% for arguments whose similarity with the corresponding parameter is greater than 0.667.

We also analyze the impact of filtering arguments associated with low-similarity parameters (Section 3.6). After the filtering, we keep 253,928 arguments, of which 100,696 (40%) have at least one alternative. Among these arguments, 55%= 55,885/100,696 are more similar to the corresponding parameters than any of their alternatives. Compared to the ratio without filtering arguments (27%), the ratio has increased by 104%=(55%-27%)/27%. One of the reasons for the increase is that most of the arguments whose similarity to their corresponding parameters is zero have been filtered out based on low-similarity parameters.

We conclude from these results that approaches that try to infer the most appropriate of all available arguments, such as code completion or anomaly detection, have a high chance to make accurate suggestions, in particular when filtering arguments based on low-similarity parameters and thresholds in minimal similarity.

# 4. APPLICATIONS

The results from Section 3 suggest several applications that exploit the similarities between arguments and parameters. In this section, we explore two such applications.

## 4.1 Anomaly Detection

We present a static analysis that detects anomalies. The main idea is to report arguments and parameters where the current argument is significantly less similar to the parameter than an alternative. The analysis helps developers in two ways. First, it reveals call sites that accidentally pass *incorrect arguments*. Based on the analysis, the developer can fix such bugs, possibly using the alternative argument suggested by the analysis. Second, the analysis reveals arguments and parameters that are correct but not appropriately named, making the code unnecessarily hard to understand and maintain. Developers should address such *renaming opportunities* by choosing identifier names that convey the semantics of the value that the identifier points to.

### 4.1.1 Approach and Implementation

Our approach for detecting incorrect arguments and renaming opportunities works as follows. For a given argument $curArg$ the analysis at first checks the corresponding parameter *par* against the set of low-similarity parameters (Section 3.6). If *par* is in this set, which suggests that it is often associated with dissimilar arguments, then the analysis ignores the current argument and does not report any warning for it. Otherwise, the analysis computes the most similar potential argument *m_alt* (Definition 3). If *m_alt* is different from $curArg$ and if the difference is above a threshold, i.e., $lexSim(m\_alt, par) - lexSim(curArg, par) \geq \beta$, then the analysis reports a warning that suggests to replace the current argument with *m_alt*, or to rename the argument or the parameter.

We implement the approach as an Eclipse plug-in that can be used in two ways. First, to check arguments incrementally and instantaneously, that is, whenever an argument is introduced or modified. In this scenario, the plug-in identifies and reports suspicious arguments immediately when the developer introduces them and suggests to the developer an alternative argument as a quick-fix. Second, to check all arguments in a project at once. In this scenario, the plug-in checks the whole application and reports all suspicious arguments, along with the source code location of each problem and suggestions for alternative arguments.

### 4.1.2 Calibration

The approach depends on a threshold $\beta$ that decides when to present warnings to the developer. In the following, we present how we calibrate this threshold using three open-source programs that are not among the subject applications of the study: *Domination* (version 1.1.1.5), *Openbravo POS* (version 2.30.2), and *Dom4j* (version 1.6.1). The applications cover different domains and are developed by different developers.

To choose a reasonable threshold, we conservatively set the threshold to $\beta = 0.4$, and apply the anomaly detection to the three applications. We manually check every reported warning and classify as a *true positive* if it points to a valid renaming opportunity or to an incorrect argument. Based on this classification, we compute the precision of the anomaly detection as follows: $Precision = \frac{\text{Number of true positives}}{\text{Number of reported warnings}}$.

For the 41 reported warnings, the similarity between arguments and parameters is discrete, and it is either 0.4, 0.5, 0.6, 0.667, or 1. We observe that the precision increases while $\beta$ increases from 0.4 to 0.667, and it decreases slightly after this point. Based on these results, we use $\beta = 0.667$ in the remaining experiments.

**1) Example from LWJGL** (commit 029fa0e)

- Signature of called method:
  `void writeVersionFile(File file, float version)`
- Incorrect method call:
  `writeVersionFile(dir, latestVersion);`
- Fix applied in commit 3656b80:
  `writeVersionFile(versionFile, latestVersion);`

**2) Example from Mondrian** (commit b583845)

- Signature of called method:
```
void putChildren(RolapMember member,
                 ArrayList children);
```
- Incorrect method call:
  `cache.putChildren(member, list);`
- Fix applied in commit c26d9f2:
  `cache.putChildren(member, children);`

**Figure 6: Examples of incorrect arguments detected by the anomaly detection.**

### 4.1.3 Evaluation

We evaluate the effectiveness of the anomaly detection, by manually identifying known problems related to incorrect arguments in the history of the subject applications and by checking whether the analysis detects these problems. To identify known problems, we use ChangeDistiller [15] to extract source code changes that affect a single argument and then manually filter those that replace an incorrect argument with a correct argument. Our methodology ensures that each considered change is indeed a bug fix. We manually inspect the commit messages and the changed code, and we keep only those changes that definitely fix a bug caused by using an incorrect argument. Most of the commit messages of the selected changes are very explicit, e.g.,"code cleanup: wrong parameter was used", "fixed bug: uploadrate is protocol+data", or "Fix for bug #44277 - correctly reference the crosstab id". We consider all applications that have a publicly accessible version control system (GIT, SVN, or CVS), which yields 51 of the 60 applications. In total, we identify 14 incorrect arguments in 11 of these applications. Figure 6 lists two example bugs. We then apply the analysis to the buggy versions of the 11 applications, manually inspect all reported anomalies, and classify each of them as incorrect argument, renaming opportunity, or false positive.

We apply the anomaly detection to the 11 subject applications with known incorrect arguments (Table 2). The approach successfully detects 6 of the 14 known incorrect arguments. Besides such 6 incorrect arguments, the approach also identifies 3 incorrect arguments that have been missed by the manual identification based on ChangeDistiller, showing that incorrect arguments are more frequent than our ChangeDistiller-based search suggests.

In addition to the 9 incorrect arguments, the analysis reports 127 renaming opportunities and 33 false positives. The average precision of the analysis, i.e., the sum of the number of incorrect arguments and renaming opportunities divided by the total number of reported anomalies, is 80%.

The detected renaming opportunities fall into four categories:

- *Abbreviations* (42/127=33%). For example, the approach warns about an argument `c` whose correspond-

**Table 2: Results of anomaly detection.**

| Application | Size (LOC) | Known Incorrect Arguments | Reported Warnings | Identified Incorrect Arguments | Identified Renaming Opportunities | False Positives | Precision |
|---|---|---|---|---|---|---|---|
| LWJGL | 32,137 | 1 | 6 | 4 | 1 | 1 | 100% |
| Mondrian | 50,003 | 1 | 4 | 1 | 2 | 1 | 75% |
| DavMail | 5,088 | 1 | 1 | 1 | 0 | 0 | 100% |
| VASSAL Engine | 98,024 | 1 | 16 | 1 | 10 | 5 | 69% |
| Vuze-Azureus (version 1.8) | 112,619 | 1 | 17 | 0 | 14 | 3 | 82% |
| Vuze-Azureus (version 1.17) | 108,043 | 1 | 14 | 1 | 10 | 3 | 73% |
| Vuze-Azureus (version 1.126) | 187,607 | 1 | 12 | 0 | 10 | 2 | 83% |
| iText | 86,952 | 1 | 54 | 0 | 43 | 11 | 80% |
| JabRef | 59,273 | 1 | 5 | 0 | 5 | 0 | 100% |
| PyDev[a] | 1,281 | 1 | 1 | 1 | 0 | 0 | 100% |
| PyDev[b] | 29,300 | 1 | 3 | 0 | 2 | 1 | 67% |
| Subsonic | 30,507 | 1 | 2 | 0 | 1 | 1 | 50% |
| JasperReports Library | 249,185 | 1 | 33 | 0 | 29 | 4 | 88% |
| Sweet Home 3D | 28,824 | 1 | 1 | 0 | 0 | 1 | 0% |
| Total | 1,078,843 | 14 | 169 | 9 | 127 | 33 | 80% |

[a] Version *e9325bcef1f1a911642b3a76cf5563753f512eaa*.
[b] Version *8fef8ba12fe8b5ff0cd37c2bb6f5c4750c18a54c*.

ing parameter is `country`, and about a name `ds` that stands for "data source".

- *Incomplete descriptions that are missing a noun (30/127 =24%).* Identifier names often consist of a noun combined with some adjunct. The approach warns about several argument names where the noun is missing, such as `missed`, which should be renamed into `missedMembers` (because the corresponding parameter is `parentMembers`), and `to_connect`, which should be renamed into `to_connect_address`.
- *Meaningless names* (17/127=13%). The approach reports argument names that reveal little or nothing about the value that the identifier refers to, such as `list` and `object`.
- *Inconsistent names* (38/127=30%). The approach reports argument names where multiple terms are used to describe a single concept, such as `file` and `module`, or `thickness` and `width`.

Most of the renaming opportunities (94%) are associated with arguments, i.e., leading to renamings of arguments, suggesting that the quality of parameter names is generally higher than that of argument names. It is reasonable in that developers usually pay more attention to parameter names because methods are expected to be called later and may be used by other developers. However, arguments are often encapsulated and hidden within methods, and thus developers rarely expect them to be read or modified by other developers.

Previous work shows that meaningful identifier names contribute to code understandability [21], and we believe that following the renaming suggestions of the analysis can greatly improve the readability of the code.

There are two main reasons for false positives reported by the approach. First, the analysis is unable to distinguish intended from unintended anomalies. For example, for the statement `bounds = new Rectangle(bounds.y, bounds.x, bounds.height, bounds.width)`, the analysis suggests swapping the last two arguments because their corresponding parameters are `width` and `height`, respectively. However, the developer intends to rotate the rectangle, i.e., the anomaly is intended. Second, the analysis currently fails to identify similarities that are obvious for a human but not for our definition of similarity, e.g., because the analysis does not tokenize names correctly or because it is unaware of irregular English plural forms. We believe that the second reason for false positives can be addressed by more sophisticated processing of names, such as Butler et al.'s method for tokenizing identifier names [7] or techniques borrowed from the natural language processing community.

## 4.2 Recommendation of Arguments

As the second application of the findings presented in Section 3, we present a name-based recommendation system that suggests arguments to a developer. Such a system can, e.g., be used as part of the code completion algorithm of an IDE, where it recommends an argument just when the developer types a method call. The key idea is to pick from the set of potential arguments the argument whose similarity with the corresponding parameter is significantly higher than any of the alternatives.

### 4.2.1 Approach

Our approach recommends arguments as follows. First, for a given argument slot, i.e., where an argument should be inserted, the approach retrieves its corresponding parameter, noted as *par*. If the name of this parameter is one of the low-similarity parameters, then the approach makes no recommendation for this argument. Otherwise, the approach collects all potential arguments (noted as $S_{pot}$): local variables, parameters of the enclosing method, invocation on methods of the enclosing class, and fields of the enclosing class. It does not consider complex expressions or literals, like `this.getAuthor().getName()` and `99`, because considering such complex expressions or literals would make the search space for potential arguments extremely large. Third, it excludes elements from $S_{pot}$ that are not type-compatible with the parameter or are not available in the slot. Excluding such elements guarantees that the recommended argument will not introduce syntactic errors. Fourth, it computes the similarity between the parameter name and

**Table 3: Results of argument recommendation.**

| Application | Size (LOC) | Recommended Arguments | Precision |
|---|---|---|---|
| Neuroph | 11,377 | 326 | 80% |
| WURFL | 10,252 | 343 | 87% |
| Json-lib | 8,055 | 122 | 92% |
| Joda-Time | 27,779 | 797 | 81% |
| Total | 57,463 | 1,588 | 83% |

the names of the collected potential arguments. Finally, if there is an argument in $S_{pot}$ whose similarity is significantly greater than others (i.e., the distance is no less than $\alpha$), the approach recommends this argument.

### 4.2.2 Calibration

To calibrate the threshold $\alpha$ and to evaluate the approach, we run the recommendation system for each argument slot in a program, i.e., for each argument position of all call sites in the program. For each slot, we compare the recommended argument against the current one. If both arguments are identical, the recommendation is considered to be *correct*. Otherwise, the recommendation is considered to be *incorrect*. Based on these data, we compute the precision of the approach as the number of correct recommendations divided by the total number of recommendations.

We calibrate the threshold $\alpha$ on three open source applications: *HtmlUnit*, *CKEditor*, and *c3p0*. We observe that while $\alpha$ is smaller than 0.5, the precision increases quickly with the increase of $\alpha$. However, after that the precision increases insignificantly with the increase of $\alpha$. Consequently, we use $\alpha = 0.5$ in the remaining experiments.

### 4.2.3 Evaluation

The approach is evaluated on four open source applications: *Neuroph*, *WURFL*, *Joda-Time*, and *Json-lib*. The evaluation results are presented in Table 3. From the table, we observe that the approach recommends 1,588 arguments with a precision of 83%.

72% of the incorrect recommendations are associated with arguments that are either complex expressions or literals, which the approach cannot recommend. Consequently, if the current argument is a complex expression or a literal, the approach fails to recommend the correct argument. 22% of the arguments in the subject applications are complex expressions or literals. Another reason for incorrect recommendations (7%) are typecasts, such as `(Map) value`. Since the approach recommends type-compatible arguments only, it cannot succeed if the actual argument is a cast expression. Excluding complex expressions, literals, and cast expressions, the precision of the recommendation is up to 96%.

Among the 1,588 recommended arguments, 1,135 (71%) are associated with inter-class invocations, i.e., the invocations (and the arguments) are out of the documents where the invoked methods are declared. The average precision for such inter-class recommendation is 84%. It is even slightly greater than that for inner-class recommendation (average precision 80%). A possible reason is that many of the inter-class invocations are associated with APIs whose parameters are often named more carefully.

## 5. THREATS TO VALIDITY

A threat to external validity is that conclusions drawn from a set of Java applications might not hold for other applications. To reduce the threat, we select the most popular open source applications from SourceForge, which yields 60 applications from various application domains. Furthermore, for RQ5, we address this threat via $k$-fold cross-validation.

For RQ4, another threat to external validity is that we manually analyze only 200 samples to investigate why some arguments are dissimilar to their corresponding parameters. To reduce this threat, we randomly select these 200 sample arguments from the population and validate the analysis results on the entire population.

The study results depend on the similarity measure that compares argument names and parameter names. We have experiments with several alternative similarity measures, but we have not found major changes in the overall results of the study. For example, when replacing Formula 1 with the string similarity-based metric from [27], the resulting accuracy in argument recommendation (82%) is almost identical to the current one (83%). In future work, it would be interesting to try additional alternative measures of similarity.

Our evaluation of the effectiveness of a name-based anomaly detection is subject to two threats to validity. First, our approach to manually identify known argument-related changes in the history of applications may not yield a representative set of argument-related bugs. We carefully inspect each of the changes that we consider as known bugs to ensure that they are indeed bugs, but we cannot ensure that we consider all such bugs in the history of these applications. Second, the classification of anomalies into incorrect arguments, renaming opportunities, and false positives is, to some degree, subjective. To reduce any potential bias, three engineers inspect each warning and must reach a consensus about its classification.

## 6. RELATED WORK

The importance of identifier names has been validated and well recognized [6, 9, 10]. As suggested by Lawrie et al.[21], there are two main sources of domain information: identifier names and comments. Because many developers do not write comments, identifier names are critical for program comprehension. A method in which names assist in reverse engineering is presented in [9]; it consist of extracting concept lattices by analyzing the identifiers names in programs.[1] Consistent naming is important and a number of approaches have been proposed to keep names consistent [4, 17, 18]. The basic idea is that a single concept within the same application should be referred to by the same name [21, 12].

A number of approaches have been proposed to calculate name similarity for identifiers. Cohen et al.[11] compare different string metrics for matching names and records. A number of approaches to tokenize identifiers have been proposed [13, 14]. Enslen et al. [13] propose an approach to split identifiers into sequences of words by mining word frequencies in source code. Butler et al.[7] propose an approach to decompose identifier names into meaningful words even if these names do not follow the camel case naming convention. Taneja et al. [30] suggest to use a synonym database to improve the accuracy of name comparison. These ap-

[1]Their paper has inspired the prefix of our title.

proaches might be used to split identifiers, and to facilitate computation of hybrid similarity [24] between identifiers. Incorporating such sophisticated approaches of computing the similarity between arguments and parameters may increase the similarities measured in our study.

Zhang et al. [32] propose an approach to recommend arguments for API usage. They mine existing code bases and build an argument usage database. For each API usage, the approach retrieves similar usage instances from the database and recommends arguments by concretizing such instances. A difference between their work and this one is that their approach depends on a large number of usage instances of the APIs. For non-API method invocation or less popular APIs, it is challenging to collect such instances.

An approach by Pradel and Gross [27, 26] relates to one of the applications of our findings, anomaly detection. They present an approach to identify problems related to the order of equally typed arguments. For each call site, they reorder equally typed arguments. If the reordered arguments match the names used at other call sites significantly better, they report a warning. Our anomaly detection differs from theirs in the following two aspects. First, they identify problems related to the order of equally typed arguments, while this work addresses arbitrary arguments. Second, they compare arguments (at a call site) with other arguments (at other call sites of the same method). In contrast, we compare arguments with corresponding parameters.

There are several approaches that identify renaming opportunities. The first category of such approaches is to identify renaming opportunities by checking identifier names against predefined rules. All names breaking such rules are presented as potential renaming opportunities. Abebe et al. [1] introduce lexicon bad smells that indicate potential lexicon construction problems. Caprile and Tonella [10] propose an approach to standardize program identifier names. First, it standardizes the lexicon (terms in identifier names). Second, it standardizes the arrangement of terms within an identifier name. Other approaches to standardize identifier names have been conducted by Lawrie et al. [19, 20] and Butler et al. [8]. Our work differs from these approaches in that it can identify renaming opportunities without requiring a set of predefined naming conventions.

The second category of work on renaming opportunities searches for inconsistent naming. Deissenboeck and Pizka [12] propose a model-based approach for identifying inconsistent naming. Based on maps between concepts and names, their approach can identify two categories of basic warnings. The first category of such warnings is given when two identifiers have identical names but different types. The second category of such warnings is given when an identifier is declared but never referenced. The first category of such warnings might suggest renaming opportunities. Our approach differs from their work in that it focuses on arguments and parameters only (arguments and parameters often refer to the same concept) and thus does not require the maps between concepts and names. It is a great advantage because it is difficult to build the maps accurately, and it usually requires domain experts. Thies and Roth [31] propose another way to identify inconsistent naming. They analyze variable assignments and identify variables that refer to the same object and are used in the same way. They suggest such variables to share the same name. Our approach differs by focusing on arguments and parameters instead of assignments.

The third category of work on renaming opportunities builds relationships between special terms and infers a set of rules, e.g., that a method that matches `contain*` should return a boolean [16]. Methods that break such rules are reported as renaming opportunities. Our approach differs from their work in that our approach identifies renaming opportunities on arguments and parameters whereas their approach identifies renaming opportunities on method names.

The fourth category of work on renaming opportunities generalize renamings conducted in the rest of the program. Once a rename refactoring is conducted manually or with tool support, Liu et al. [22] recommend to rename closely related software entities whose names are similar to that of the renamed entity. The approach proposed in this paper differs from that in [22] in that it does not depend on conducted renamings.

Arnaoudova et al. [5] analyze and classify identifier renamings, e.g., by comparing the original and the new name using an ontological database [23]. Future work may improve our similarity measure using such ontological databases. Statistical language models extracted from a corpus of code have been used to automatically suggest identifier names for variable names [2], as well as method and class names [3]. JSNice [28] predict names of local variables in JavaScript applications.

# 7. CONCLUSIONS AND FUTURE WORK

This paper presents the first in-depth empirical study of similarities between the names of arguments and parameters of methods. Our results show that identifier names of arguments and parameters are often similar to each other, that dissimilar names can be filtered based on low-similarity parameters inferred from a set of sample programs, and that many arguments are the most similar to the corresponding parameter of all possible arguments that are available in the scope of the method call. As applications of our findings, we present an anomaly detection technique that identifies renaming opportunities and potentially incorrect arguments, as well as a recommendation system that suggests arguments to a developer who is typing a method call.

The broader impact of our work is to show that identifier names are a rich source of information that can provide otherwise missing information to program analyses. We expect our results to encourage future research on name-based program analyses, which will complement existing program analyses for several software engineering tasks. For example, names may improve code completion algorithms, support the generation of documentation, and support fault localization.

## Acknowledgments

This research is supported by the National Natural Science Foundation of China (61272169, 61472034), the Program for New Century Excellent Talents in University (NCET-13-0041), the Beijing Higher Education Young Elite Teacher Project (YETP1183), the German Federal Ministry of Education and Research (EC SPRIDE and CRISP), and by the German Research Foundation within the Emmy Noether Project “ConcSys”

# 8. REFERENCES


[1] S. Abebe, S. Haiduc, P. Tonella, and A. Marcus. Lexicon bad smells in software. In *WCRE*, pages 95–99, Oct 2009.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *FSE*, pages 281–293, 2014.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *ESEC/FSE*, pages 38–49, 2015.

[4] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON*, page 4, 1998.

[5] V. Arnaoudova, L. Eshkevari, M. Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, May 2014.

[6] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *CSMR*, pages 156–165, 2010.

[7] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the tokenisation of identifier names. In *ECOOP*, pages 130–154. 2011.

[8] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Mining Java class naming conventions. In *ICSM*, pages 93–102, Sept 2011.

[9] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *WCRE*, pages 112–122, 1999.

[10] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, pages 97–107, 2000.

[11] W. W. Cohen, P. D. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Workshop on Information Integration on the Web*, pages 73–78, 2003.

[12] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[13] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *MSR*, pages 71–80, 2009.

[14] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *International Conference on Software Engineering and Applications (SEA)*, pages 156–165, 2006.

[15] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, pages 725–743, 2007.

[16] E. W. Høst and B. M. Østvold. Debugging method names. In *ECOOP*, pages 294–317, 2009.

[17] P. Jablonski and D. Hou. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Workshop on Eclipse Technology eXchange*, pages 16–20, 2007.

[18] P. Jablonski and D. Hou. Renaming parts of identifiers consistently within code clones. In *ICPC*, pages 38–39, 2010.

[19] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *ICSM*, pages 113–122, 2011.

[20] D. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *WCRE*, pages 3–12, 2010.

[21] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *ICPC*, pages 3–12, 2006.

[22] H. Liu, Q. Liu, Y. Liu, and Z. Wang. Identifying renaming opportunities by expanding conducted rename refactorings. *IEEE Transactions on Software Engineering*, (99):1–1, 2015.

[23] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[24] A. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Workshop on data mining and knowledge discovery*, pages 267–270, 1997.

[25] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *ICSE*, pages 815–825, 2012.

[26] M. Pradel and T. R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *ISSTA*, pages 232–242, 2011.

[27] M. Pradel and T. R. Gross. Name-based analysis of equally typed method arguments. *IEEE Transactions on Software Engineering*, 39(8):1127–1143, 2013.

[28] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *POPL*, pages 111–124, 2015.

[29] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, pages 419–428, 2014.

[30] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE*, pages 377–380, 2007.

[31] A. Thies and C. Roth. Recommending rename refactorings. In *Workshop on Recommendation Systems for Software Engineering*, pages 1–5, 2010.

[32] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical API usage. In *ICSE*, pages 826–836, 2012.

[33] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, pages 307–318, 2009.