Empirical software engineering manuscript No.
(will be inserted by the editor)

Lexical Similarity between Argument and Parameter Names: An Empirical Study

Guangjie Li · Hui Liu · Yuting Wu · Cristian Alexandru Staicu · Michael Pradel

Received: date / Accepted: date

Abstract Identifier names chosen by programmers convey rich semantic information that could be exploited to enhance program analysis for various software engineering tasks, e.g., anomaly detection, argument recommendation, and code completion. However, such information has not yet been fully exploited, and little is known about its properties. In this paper, we carry out an empirical study to investigate the lexical similarity between argument and parameter names. We collect and analyze arguments and parameters from 127 Java applications and 30 open-source C applications. Based on the analysis, we observe that (1) 87% of Java arguments are either identical to or completely different from their corresponding parameters. (2) Short parameter names and generic collection-related parameter names account for 79% of the dissimilarity between argument and parameter names. (3) Filtering out a set of low-similarity parameter names based on sample applications helps pruning such names in other applications. (4) 82% of API arguments are completely different from their corresponding parameters. (5) 43% of Java arguments are more similar to the corresponding parameter than any alternative arguments available for the call sites. (5) Switching from one similarity metric to other metrics, e.g., *Levenshtein distance-based* similarity, has little influence on the conclusions. (6) These findings apply to both Java and C applications. Evaluation results of two name-based applications, i.e., argument

Guangjie Li, Hui Liu,and Yuting Wu
School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
E-mail: 3120150428@bit.edu.cn,liuhui08@bit.edu.cn,wuyuting@bit.edu.cn

Cristian-Alexandru Staicu, Michael Pradel
Department of Computer Science, TU Darmstadt, Germany
E-mail: cris.staicu@gmail.com,michael@binaervarianz.de

Bing Xie
Software Engineering Institute, Peking University, China
E-mail: xiebing@pku.edu.cn

recommendation and anomaly detection, suggest that the similarity between argument and parameter names may facilitate simple but effective approaches for argument-related software engineering tasks.

Keywords Empirical study · program analysis · identifier · static analysis · argument · lexical similarity

1 Introduction

Identifier names provided by programmers often convey rich information about the semantics of a program(Lawrie et al 2006). However, most existing program analyses do not make full use of such information. As a result, there is little difference between analyzing a program with meaningful identifier names and analyzing an equivalent program where all identifiers are consistently replaced with arbitrary strings. Ignoring the valuable information conveyed in identifier names may limit potential analysis on program. To exploit this information, recent work uses identifier names to infer specifications of APIs (Zhong et al 2009; Pandita et al 2012), to identify mismatches between a method name and the method's implementation (Høst and Østvold 2009a), to synthesize code completions (Raychev et al 2014), to predict syntactic and semantic properties of programs (Raychev et al 2015), to detect incorrectly ordered method arguments of the same type (Pradel and Gross 2011, 2013), and to generate comments for parameters (Sridhara et al 2011).

However, these approaches provide few insights about the properties of identifier names in real-world software. Does there exist some kind of similarity between identifier names that refer to semantically related values? Can we predict which identifier name will be used as argument next time by the developers? Can we automatically identify bugs in programs just based on the identifier names? Answering these questions may pave the way for name-based analyses that complement existing approaches. For example, one could exploit the similarity of names to automatically complete pieces of code, to infer documentation from names, to warn developers about anomalies that may correspond to code worth changing, or to recommend concise and consistent new names.

This paper studies the lexical similarity between arguments and parameters of methods. Note that we refer to "argument" as values passed to a method at a call site, and refer to "parameter" as the formal parameter in the method's definition. The names of arguments and parameters are important to convey their semantics. For example, consider a method `substring` with two integer parameters. How does a developer who calls the method know the meaning of these parameters? The identifier names, say `beginIndex` and `endIndex`, are crucial to understand the semantics. Specifically, the developer is likely to match these parameters names with names of possible arguments she may pass and, e.g., pass two arguments called `begin` and `end`.

We hypothesize that an argument and its corresponding parameter are often similar because their names often refer to the same concept. To evaluate

this hypothesis, we conduct an empirical study with 754,710 arguments from 127 Java applications, and 345,602 arguments from 30 C applications. The main findings of the study on Java arguments are introduced as following:

• The distribution of the lexical similarity has a U-shape: arguments are either identical to or completely different from their corresponding parameters. On one side, many arguments (43%) exactly match their corresponding parameter names. On the other side, many arguments (44%) share no common terms with their corresponding parameter names. Such a U-shape distribution may facilitate the filtration of highly similar (or highly dissimilar) pairs of argument-parameter names, and thus facilitate name-based argument analyses, e.g., detection of incorrect arguments.

• A large number of the parameter names are short, and such short names are one of the main reasons for the dissimilarity between arguments and parameters. 80% of the parameter names contain no more than 8 characters, 99% of the parameter names are composed of at most 3 terms. Meanwhile, 43% of the dissimilarity between argument and parameter names is caused by short (no more than 3 characters) parameter names. This finding suggests that describing parameters with more terms improves not only the readability of source code, but also the lexical similarity between arguments and parameters.

• Most (81%) dissimilar pairs of argument and parameter names are due to a small set of parameters that occur again and again across programs. 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.

• Most (82%) arguments used in API invocations, where the called methods are defined by the JDK, are dissimilar to their corresponding parameters. In contrast, only 31% of arguments associated with non-API parameters are dissimilar to their corresponding parameters. This finding suggests that to improve the performance of name-based argument analyses, API arguments and non-API arguments should be distinguished and handled in different ways.

• Switching from term-based similarity metrics to other similarity metrics, e.g., Levenshtein distance-based similarity metrics or JaroWinkler-based similarity metrics, has little influence on the conclusions drawn in preceding paragraphs. The term-based similarity (Formula 1 in Section 2.2) counts the comment terms whereas *Levenshtein distance-based* similarity (Formula 8 in Section 3.8.1) counts the common characters, and *JaroWinkler-based* similarity (Formula 9 in Section 3.8.1) is based on the number and order of the common characters. This finding suggests that name-based argument analyses may work well with diverse types of similarity metrics.

To investigate whether the conclusions hold on other programming languages (especially non-object-oriented programming languages), we repeat the study with 345,602 C arguments from 30 open-source applications. Results suggest that the conclusions drawn on Java hold on C as well.

This article significantly extends a previously presented conference paper (Liu et al 2016) in two ways:

- We extend the scope of the study by introducing four additional research questions. The added questions consider the types of arguments (RQ8), distinguish between API and non-API arguments (RQ9), apply different string similarity metrics (RQ10), and compare different programming languages (RQ11). The investigation leads to new findings, e.g., that API parameters are more likely to be dissimilar to arguments than other parameters, which provide guidance for applications of name similarities.
- We increase the generalizability of the results by considering a larger and more diverse data set. In this paper, the number of involved Java applications increases from 60 into 127. Besides that, in this paper we also analyze 30 open-source C applications from GitHub that contain 345,602 arguments. In total, the number of arguments increases by 81%. This extended data set enables us to assess to what extent the previous findings (Liu et al 2016) apply to other applications and to another programming language.

Based on the results of our study we explore two applications of name-based analysis: anomaly detection to find programming errors and code completion. As a real-world example of the impact that our study has, we refer to an effort at Google: Our previous conference paper (Liu et al 2016) motivated the company to adopt a name-based bug detector that has detected several thousands of naming-related bugs (Rice et al 2017).

2 Setup

2.1 Research Questions

- *RQ1*: How similar are argument names to their corresponding parameter names? Answering this question helps to decide whether it is worthy to explore similarities between argument and parameter names.
- *RQ2*: How pervasive are non-variable arguments, e.g., call expressions, field accesses, and *this* expressions, and how do their properties differ from variable arguments?
- *RQ3*: How long are argument and parameter names, and how is the length of argument and parameter names related to the similarity between argument and parameter names? Answering this question may help estimate how much confidence one can have in similarities between names of particular lengths.
- *RQ4*: How often does an overriding method change the parameter names? Answering this question helps to estimate how often comparing arguments to parameters of the statically resolved call target is sufficient, and how often considering the dynamic call target would yield different results.
- *RQ5*: What kinds of parameters are more likely to be dissimilar with corresponding arguments? Answering this question helps applications that exploit name similarities to distinguish or ignore particular kinds of parameters.

- *RQ6*: If parameters with specific names, e.g., `index`, are frequently dissimilar from their arguments in sample applications, will they be dissimilar to arguments in other applications? If yes, can we filter out these parameters and build up a set of *low-similarity parameters* that are likely to be assigned to dissimilar arguments? Answering this question helps to improve 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.
- *RQ7*: How often is the argument chosen by developers more similar to the corresponding parameter than any of its potential alternatives? Potential alternatives are any available identifier names in the current scope that can be used as argument and do not introduce syntactical errors or type errors. Answering this question helps to estimate the accuracy of name-based analyses, i.e., name-based anomaly detection and name-based argument recommendation.
- *RQ8*: As far as the lexical similarity between argument and parameter names is concerned, is there any essential difference between primitive and non-primitive parameters? Answering this question indicates whether ignoring a subset of parameters can improve the accuracy of name-based analyses.
- *RQ9*: As far as the similarity between argument and parameter names is concerned, is there any essential difference between API and non-API invocations? Answering this question helps to improve the accuracy of name-based recommendation by ignoring API or non-API parameters.
- *RQ10*: To what degree do these results depend on the similarity metrics used to compare arguments and parameters?
- *RQ11*: As far as lexical similarity between argument and parameter names is concerned, do the conclusions drawn on Java applications hold for C applications as well? Answering this question helps to estimate whether the conclusions drawn on Java (object-oriented programming language) applications hold for other languages (especially procedure-oriented programming language).

2.2 Experimental Setup

To empirically investigate the lexical similarity between argument and parameter names, we analyze 125 open-source Java applications, 2 closed-source Java applications, and 30 open-source C applications.

Different expressions may be passed as arguments, e.g., variables, literals, method invocations and complex expressions (like *length+width*). To simplify the analysis, in the first round of the study we consider the most popular type of arguments only: variables. Variables account for around 80% of arguments. After that, we also analyze the other types of arguments in Section 3.2 and validate whether conclusions drawn on the first type of arguments (variables) hold for other types of arguments as well.

To compare argument names against parameter names, the analysis retrieves the called method by static analysis. In some cases, knowing which method will be called is a hard problem for static analysis because of the runtime polymorphism in Java, e.g., a particular call site may call a particular implementation of an overridden method. However, results in Section 3.4 suggest that in most cases overriding methods have the same parameter names as the overridden methods. Consequently, resolving method invocations with static analysis is sufficiently accurate in most cases. Based on the extracted argument and parameter names, we compute the lexical similarity of two names as follows. First, we decompose each argument name arg and each parameter name par into a list of terms, noted as $terms(arg)$ and $terms(par)$ respectively. The decomposition is based on underscores and capital letters, assuming that the name follows the popular camel case or snake case naming convention. Second, we measure the *term-based* similarity between argument arg and parameter par as follows:

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

$comterms(arg, par)$ is a subsequence of $terms(arg)$, where each term in the subsequence appears in $terms(par)$ (case insensitive). For example, if

$$arg = \text{“columnNameByCvsFileName”} \tag{2}$$
$$par = \text{“columnName”} \tag{3}$$

we have:

$$comterms(arg, par) = < column, name, name > \tag{4}$$
$$comterms(par, arg) = < column, name > \tag{5}$$
$$tSim(arg, par) = \frac{3+2}{6+2} = 0.625 \tag{6}$$

Note that the *term-based* similarity is highly similar to the well-known Jaccard similarity (Cohen et al 2003). The major difference is that Jaccard ignores repeating terms. For the example in the preceding paragraph, Jaccard similarity is computed as follows:

$$\begin{aligned} jacSim(arg, par) &= \frac{|terms(arg) \bigcap terms(par)|}{|terms(arg) \bigcup terms(par)|} \\ &= 2/5 \\ &= 0.4 \end{aligned} \tag{7}$$

Alternative lexical similarity metrics are also employed and compared in Section 3.8.1 to investigate the influence of different lexical similarity metrics.

2.3 Subject Applications

The subject applications, made available at (dat 2017-9-23), are composed of three groups. The first group are 125 open-source Java applications, including the 60 applications analyzed in our previous paper (Liu et al 2016). These applications are the most popular Java applications from SourceForge (sou 2016-5-8)[1] that have at least 5 releases. The size of such Java applications varies from 935 to 717,732 lines of source code (LOC, excluding blank lines and comment lines). In total, the Java applications are composed of 18,657,548 lines of source code. From these applications, we extract 751,007 arguments.

The second group are 30 open-source C applications. To validate whether the conclusions drawn from Java applications hold for other languages, we retrieve the most popular 30 open-source C applications that have at least 5 releases from GitHub(git 2016-8-2)[2] for validation. The size of such C applications varies from 3,727 to 2,060,180 lines of source code (LOC, excluding blank lines and comment lines). In total, the C applications are composed of 7,890,865 lines of source code. From these applications, we extract 345,602 arguments. We choose C applications to validate the conclusions drawn on Java applications because C is one of the most popular procedure-oriented programming languages, whereas Java is one of the most popular object-oriented programming languages. Comparing results on object-oriented and procedure-oriented programming languages may help to reveal the impact of programming languages on the similarity between argument and parameter names.

The third group are two commercial applications. To validate whether conclusions drawn on open-source applications hold as well on closed-source applications, we analyze arguments from two commercial applications. One of the applications is an instant refactoring framework (called CSA1), the other is an Eclipse plug-in to detect software defects (called CSA2). The two applications are composed of 140,516 lines of source code (LOC, excluding blank lines and comment lines). From these applications, we extract 3,703 arguments.

The distribution of the similarity between arguments and parameters may vary according to the types of parameters (arguments). In this paper, we will investigate whether Java primitive type arguments or Java API arguments have specific characters concerning the similarity between arguments and parameters. Java primitive type arguments are those whose types are one of the Java basic data types, including *byte, char, boolean, short, int, long, float, double* and *string*. Such arguments account for 36% (=269,485/754,710) of the Java arguments. It is possible that conclusions drawn on non-primitive arguments may not hold for primitive arguments.

[1] May 8, 2016
[2] August 2, 2016

Java API arguments are arguments associated with API invocations[3]. API arguments account for 26% (=199,441/754,710) of the Java arguments. APIs are designed to be widely used by different developers in different applications, whereas non-API methods are designed to be used by specific developers within the specific application. Consequently, it is possible that conclusions drawn on non-API arguments may not hold for API arguments.

3 Results

3.1 Distribution of Lexical Similarity

The distribution of the similarity between arguments and their corresponding parameters is presented in Fig. 1. From this figure, we observe that the distribution is a U-shape bimodal distribution: the similarity is either extremely high or extremely low. 87% of Java arguments, 84% of Java primitive type arguments, 92% of Java API arguments, and 92% of C arguments are exactly identical to or completely different from their corresponding parameters.

One of the reasons for the bimodal distribution is that arguments and their corresponding parameters often semantically refer to the same concept. Consequently, it is highly possible that they are described with the same terms. From Fig. 1, we observe that a large percentage (45%) of arguments are identical to their corresponding parameters. Another reason for the bimodal distribution is that, developers often use abbreviations (especially with a single letter) for convenient to stand for full names. The result of an empirical study suggests that single-letter abbreviations make up 9-20% of the variable names(Beniamini et al 2017). Although such abbreviations are semantically similar to their full expansions, they are lexically dissimilar at all. For example, "i" abbreviates for "index", whereas it is lexically dissimilar to "index" at all. Consequently, the large amount of abbreviations are destined to a large percentage of dissimilarity with 0. According to our analysis, 56% of arguments and 73% of parameters are named with a single term. The similarity between such arguments and parameters is either 1 (the only term is common) or 0 (no comment term at all). Consequently, we observe a U-shape distribution. Fig. 2 presents the distribution of the length of argument-parameter pairs, as well as their common terms. The horizontal axis presents not only the similarity between argument and parameter names, but also the denominator of the similarity, i.e., the total length of argument and parameter names $|terms(arg)| + |terms(par)|$ (in our case the maximum length of argument-parameter pairs is up to 20 total terms). From this figure for around 50% of argument-parameter pairs, both argument and parameter names are composed of a single term (i.e., the total length is 2). As a result, the similarity between the arguments and their corresponding parameters is either 1 (59% of the cases) or 0 (41% of the cases).

[3] In this paper, we consider invocations of JDK APIs only.

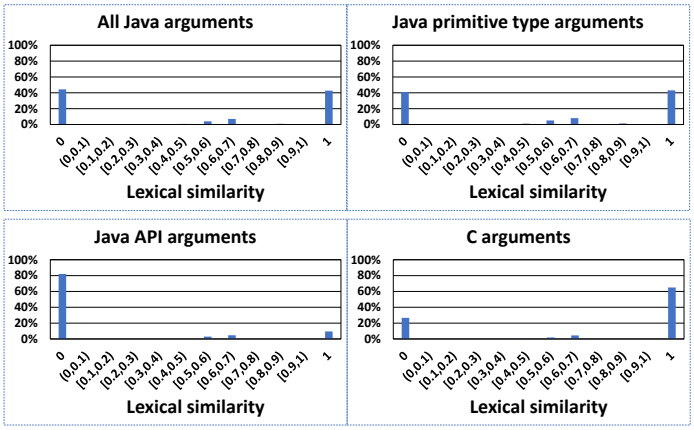

Fig. 1: Distribution of lexical similarity between arguments and parameters.

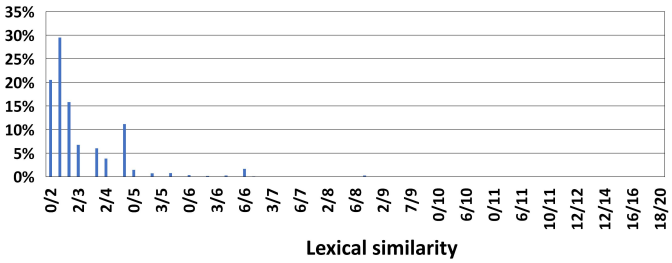

Fig. 2: Distribution of lexical similarity and total length of argument-parameter names.

The general conclusion that argument-parameter names are either extremely similar or extremely dissimilar holds for different kinds of arguments, including all Java arguments, Java primitive type arguments, Java API arguments, and C arguments. However, we also observe some differences between their similarity distributions. In the following paragraph, we discuss such differences.

First, we observe an obvious difference between the two subgraphs on the left of Fig. 1 : more than 80% of the API arguments are completely different from their corresponding parameters whereas only 44% of the Java arguments (including both API and non-API arguments) are completely different from their corresponding parameters. To investigate the reasons for the difference, we retrieve the top 10 API parameter names and the top 10 API packages that are most frequently associated with dissimilar arguments. The top 10 API parameter names are presented in Fig. 3. The horizontal axis represents the parameter names. The vertical axis represents the number of dissimilar ar-

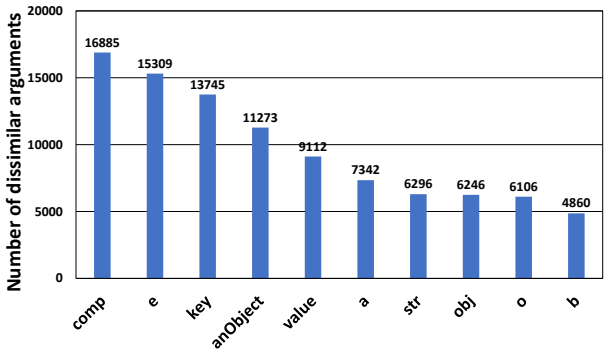

Fig. 3: Top 10 API parameter names most frequently associated with dissimilar argument names.

guments associated with the parameter name. Overall, 59% (=97,174/163,762) of the dissimilar API arguments are assigned to the top 10 dissimilar API parameter names. From this figure, we observe that the parameter name "comp" (abbreviation of "component") is most frequently associated with dissimilar API arguments. Approximately 10% (=16,885/163,762) of the dissimilar API arguments are associated with this parameter name. We also observe that the most common argument names associated with "comp" are "label", "jPanel$n$" (where $n = 1, 2 \ldots$), "ButtonPanel", and "okButton". Developers employ such argument names because such arguments belong to specific types of components, e.g., *label*, *panel* or *button*. However, API designers employ the parameter name "comp" (or "component") because they expect the API to handle all kinds of components in the same universal way. As a result, the arguments are lexically mismatched with the API parameters. The same is true for other popular API parameter names, e.g., "e" (event), "key", "value", and "obj" (object).

The top 10 API packages that are most frequently associated with dissimilar arguments are presented in Fig. 4. From this figure, we observe that "java.util" is the package most frequently associated with dissimilar arguments. Approximately 34% (=55,538/163,762) of the dissimilar API arguments are associated with methods from this package. It is natural because this package is composed of generic algorithms and tools [4]. The more generic the algorithms (methods) are, the more likely that the parameter names are generic. The analysis results may suggest that name-based argument analysis, e.g., argument commendation and detection of incorrect arguments, may improve their accuracy significantly by simply ignoring such API packages.

Second, although the distribution of the similarity between arguments and parameters in C programs has a U-shape as well, the distribution is dominated by a similarity of 1. In other words, similar argument-parameter pairs

[4] http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html

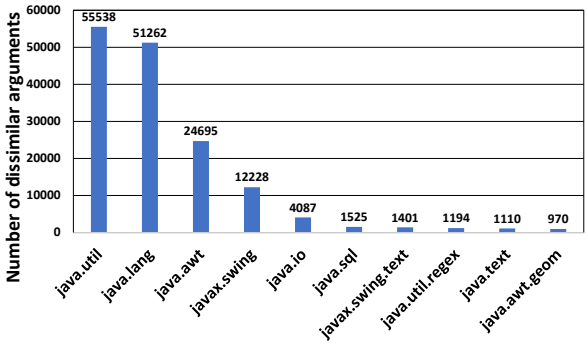

Fig. 4: Top 10 API packages most frequently associated with dissimilar argument names.

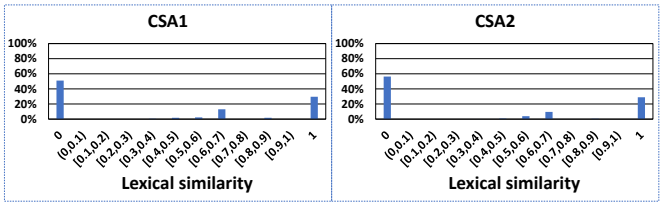

Fig. 5: Distribution of lexical similarity between arguments and parameters (closed-source applications).

are more prevalent in C. To investigate the reasons, we manually inspected 200 C arguments and 200 Java arguments that are completely different from their corresponding parameters. We find that collection-related parameter names, i.e., *index*, *item*, *key*, *value*, are common in Java whereas they are significantly less common in C. Such names account for 8% (=60,321/754,710) of Java parameter names and 1.6% (=5,468/345,602) of C parameter names, respectively. As analyzed in Section 3.5, such parameter names often lead to lexical mismatch between argument and parameter names. Consequently unpopularity of such names improves the overall similarity between C argument and parameter names. Another possible reason for greater similarity between C argument and parameter names is that single character parameter names are significantly less popular in C: They account for 10% and 24% of C and Java parameters, respectively. As analyzed in Section 3.5, such single character parameter names often lead to lexical mismatch between argument and parameter names.

To validate whether the conclusions drawn from open-source applications hold for closed-source applications, we investigate the similarity distribution of two commercial applications. The results are presented in Fig. 5. From this figure, we observe that the distribution is a U-shape as well.

To investigate the relationship between lexical similarity and the semantical similarity between argument and parameter names, we carry out the following case study. First, we randomly sampled 50 pairs of lexically similar argument-parameter names whose similarity is greater than 0.5, and 50 pairs of lexically dissimilar argument-parameter names whose similarity is less than 0.5. Second, for each pair of the argument-parameter names, we presented it to five software engineers, and asked them to tell whether the names are semantically very similar, similar, or dissimilar on the first round. Third, when disagreement appears, all of the participants were asked to look into the code and then retell the semantic similarity of the name pair on the second round. Finally, for the argument-parameter pairs that participants still can not reach consensus on, they would be discarded on the final round. In total 92 pairs of names were given the same choice in the very first round, 6 pairs reached consensus in the second round discussion, whereas 2 pairs of dissimilar argument-parameter names were finally discarded. Results of the case study suggest that there is strong correlation between lexical similarity and semantic similarity. On the one hand, up to 96% (=48/50) of the lexically similar pairs are semantically very similar, and others (4%) are similar. On the other hand, 79% (=38/48) of the lexically dissimilar argument-parameter pairs are semantically dissimilar as well. However, we also notice that up to 24% of the lexically dissimilar argument-parameter pairs are semantically similar (13%=6/48) or very similar (8%=4/48). One of the reasons for the mismatch between lexical similarity and semantical similarity is the generality of parameter names, accounting for 60% (=6/10) of the mismatch. As analyzed in the preceding paragraphs, parameter names (especially API parameters) like "comp" are often generic whereas the arguments like "jPanel1" are specific and concrete. Another reason for the mismatch are abbreviations (e.g., "t" VS. "time"), accounting for 30% (=3/10) of the mismatch. The third reason for the mismatch are (near) synonyms, such as "start" and "begin". It accounts for 10% (=1/10) of the mismatch. Based on the analysis, we conclude that there is strong correlation between lexical similarity and semantic similarity of argument-parameter names although mismatch may appear for different reasons.

As a conclusion of the analysis in preceding paragraphs, arguments are often either highly similar or highly dissimilar to their corresponding parameters (RQ1). This conclusion holds for Java arguments, primitive arguments (RQ8), API arguments (RQ9), and C arguments (RQ11). The finding may facilitate the filtration of highly similar (or highly dissimilar) pairs of argument-parameter names, and thus facilitate name-based argument analyses, e.g., detection of incorrect arguments.

3.2 Non-Variable Arguments

In the preceding section, only the most popular type of arguments are analyzed: variables. In this section, we investigate how frequent other forms of

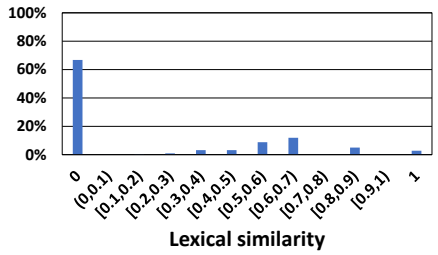

Fig. 6: Distribution of Lexical Similarity between Non-Variable Arguments and Parameters

arguments are, i.e., method invocations, field accesses, and *this* expressions. For convenience, we call such kinds of arguments *non-variable arguments*. We also investigate the distribution of the similarity between non-variable arguments and their corresponding parameters, and compare the distribution against that of variable arguments.

Identifier names of the non-variable arguments are extracted as follows:

- For a call expression, the argument name is the name of the called method. We ignore the base expression. For example, if the return value of a method invocation *stack.push*() is passed as an argument, then the argument name is *push*.
- For a field access, the argument name is the name of the field. We ignore the base expression. For example, if the argument is *stack.size*, then *size* is taken as the argument name.
- For the *this* expression, the argument name is the name of the class of which *this* is an instance. For example, in a method invocation *display(this)*, if the argument *this* refers to an instance of class *Stack*, then the argument name is *Stack*.

In total, we extract 182,828 non-variable arguments including 136,886 call expressions, 18,662 field accesses, and 27,280 *this* expressions. Such non-variable arguments account for 20% (=182,828/(182,828+754,710)) of all arguments.

The distribution of lexical similarity between non-variable arguments and parameters is presented in Fig. 6. From this figure we observe that compared to variable arguments, non-variable arguments are more likely to be lexically dissimilar with corresponding parameters. Up to 67% of the non-variable arguments are completely different from their corresponding parameters. One of the reasons for the extensive dissimilarity is that *this* expressions are often lexically dissimilar with parameters. For 89% of such arguments, they are completely different from their corresponding parameters, i.e., the lexical similarity is zero. Since it is not clear what the name of the instance (*this*) would be by static source code analysis, we take the class name (type) of the instance as the identifier name. As a result, the class name (specifying a category of instances) is often dissimilar with the parameter name (specifying a

specify instance). Another reason for the extensive dissimilarity is that call expressions often contain verbs whereas parameter names rarely contain verbs, which leads to massive mismatch between argument (call expressions) and parameter names.

As a conclusion of the analysis, non-variable arguments account for 20% of arguments, and they are less similar to their parameters (RQ2).

3.3 Length of Names and Its Impact

3.3.1 Length of Names

To investigate the question how long argument and parameter names are, we measure such names in characters and terms, respectively. Results are presented in Fig. 7.

From this figure, we observe that most arguments are composed of no more than 10 characters and most parameters are composed of no more than 8 characters. For example, 68% of the Java argument names contain 3 to 10 characters, and 80% of the parameter names contain no more than 8 characters. We also observe that most argument (and parameter) names are composed of no more than 3 terms, whereas single-term names account for a large amount of them. For example, 99% of Java parameter names, 98% of Java argument names, almost 100% of C parameter names, and 95% of C argument names are composed of no more than 3 terms. Single-term names account for 56% of Java argument names, 73% of Java parameter names, 66% of C argument names, and 78% of C parameter names, respectively.

We also observe that the average length of argument names is greater than that of parameter names. For example, the average length of Java argument names is 7.4 characters or 1.6 terms, whereas the average length of Java parameter names is 5.8 characters or 1.3 terms. One of the reasons for the difference may be that single-character names account for 13% of parameter names, whereas they account for only 6% of argument names.

3.3.2 Correlation between Length and Similarity

To investigate the question how the length of names is related to the lexical similarity between argument and parameter names, we also investigate their correlation.

Analysis results are presented in Fig. 9. From this figure we observe that the lexical similarity decreases when the length of argument names increases. In contrast, the lexical similarity increases when the length of Java parameter names increases whereas it decreases when the length of C parameter names increases.

However, the correlation between the length of names (arguments or parameters) and the similarity is weak. For example, the correlation coefficient is as low as -0.048 (arguments) and 0.267 (parameters) for Java arguments. It

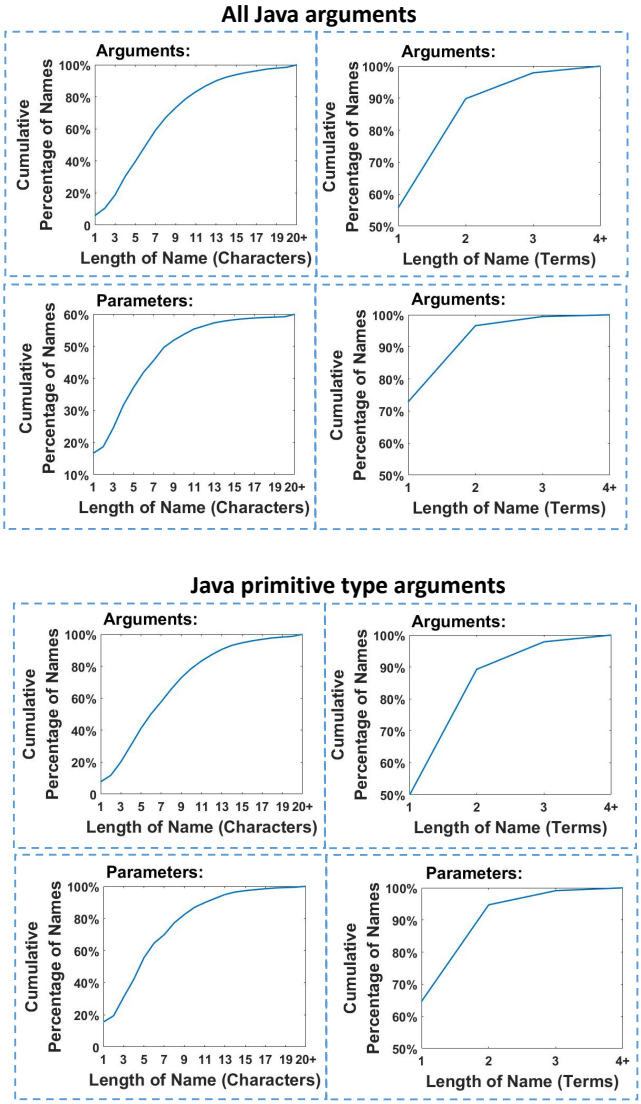

Fig. 7: Length of argument and parameter names (cumulative distribution).

even decreases to -0.025 (parameters) and -0.324 (arguments) for C arguments. 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 also calculate the *p-value* associated with such coefficients. The *p-value* is the probability, calculated un-

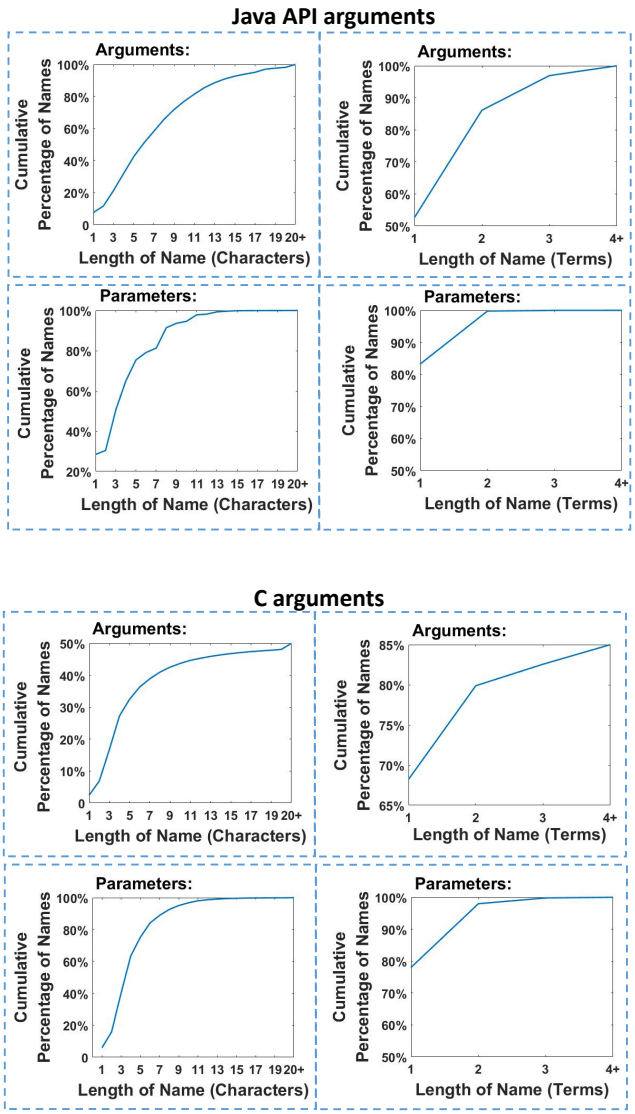

Fig. 8: Length of argument and parameter names (cumulative distribution).

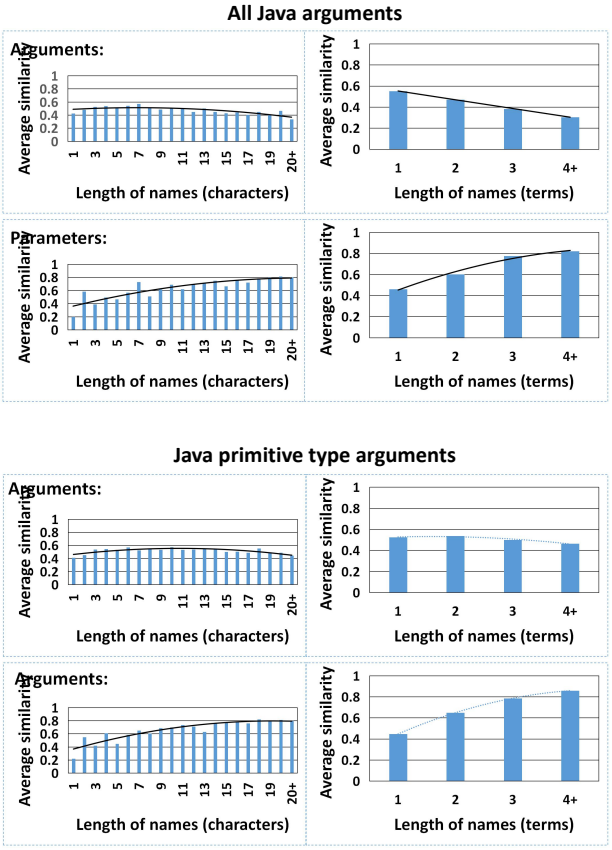

Fig. 9: Correlation between length of names and average similarity (with polynomial trendlines).

der the null hypothesis (i.e., there is no correlation between length of names and their similarity), of having outcome as extreme as the observed value in the sample (pva 2017). The p-values for all of the coefficients turn out to be zero, suggesting that it is impossible to see such observations if the similarity is not related to length of names. Consequently, although the correlation is weak, the similarity is related to the length of names.

We also observe that the correlation between the similarity and the length of C names is even weaker compared to that of Java names. One of the possible reasons is that most C names distribute within a narrow length. For example, 41% of C argument names and 48% of C parameter names are composed of 3-4 characters.

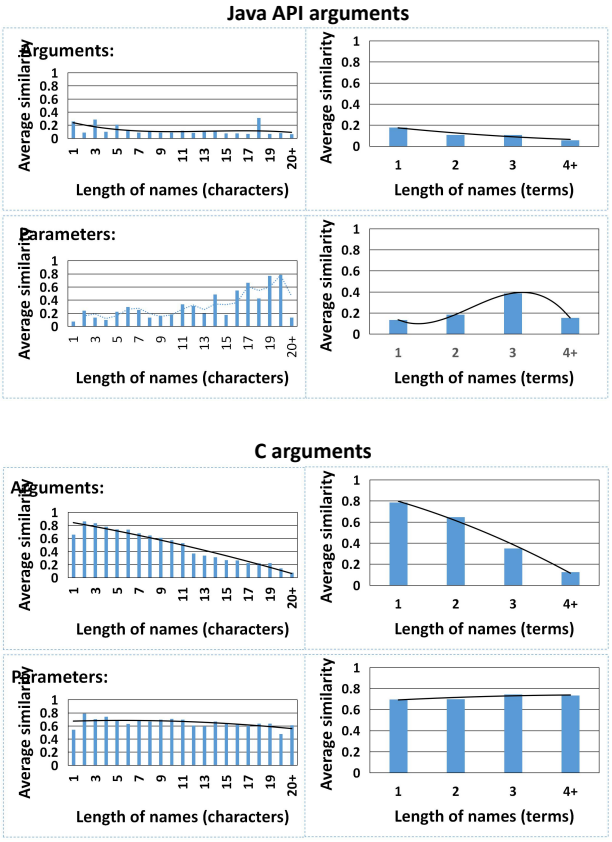

Fig. 10: Correlation between length of names and average similarity (with polynomial trendlines).

We conclude from these results that the similarity between Java arguments and parameters increases with the length of parameter names increasing (RQ3). This finding suggests that describing parameters with more terms improves not only the readability of source code, but also the lexical similarity between arguments and parameters.

3.4 Impact of Polymorphism on Results of Parameter Names

We statically resolve Java method calls based on well-known JDT [5] of Eclipse because static resolution is simple and straightforward. However, knowing which of the methods that will be called is a hard problem for such static

[5] http://www.eclipse.org/jdt/

```
1.  public class Tree{
2.      public void display(String tree){}
3.  }
4.  public class BinaryTree extends Tree{
5.      public void display(String binary){}
6.  }
7.  public class TreeVisitor{
8.      String name="***";
9.      public void traverse(Tree treeNode){
10.          treeNode.display(name);
11.     }
12.  }
```

Fig. 11: Overriding methods and their static resolution.

analysis due to runtime polymorphism in Java.For example, in Fig. 11, it is challenging for statical source code analysis tools to tell which method will be called by the invocation statement *treeNode.display(name)* (Line 10). If the enclosing method *traverse* is invoked with an argument of type *Tree*, method *display(String tree)* (Line 2) would be invoked. As a result, the corresponding parameter of method invocation *treeNode.display(name)* is "tree". However, it is also possible to invoke the enclosing method *traverse* with an argument of type *BinaryTree*, a subtype of *Tree*. In this case, method *display(String binary)* (Line 5) would be invoked, and the corresponding parameter is "binary" instead of "tree". However, JDT will always resolve the *treeNode* in Line 9 as type *Tree* and the *treeNode.display(name)* in Line 10 as invocation to method *display(String tree)* (Line 2). Consequently, our static analysis always returns "tree" as the parameter name even if in fact the overriding method *display(String binary)* is invoked.

To investigate how often such error-prone resolution may happen, we analyze the parameters of overriding and overridden methods in the subject applications. Analysis results suggest that out of the 748,550 methods with parameters, only 8,757 methods are overridden by 30,913 overriding methods. Such overridden and overriding methods account for 1.2% (=8,757/748,550) and 4% (=30,913/748,550) of the analyzed Java methods (excluding methods without parameters as well), respectively. Analysis results also suggest that up to 92% (=28,380/30,913) of the overriding methods have exactly the same parameter list (including parameter names) as the methods they override. As a result, parameters of the overriding methods often (95% of the cases) have the same names as they are defined in the corresponding overridden methods.

We conclude from these results in most cases overriding methods have the same parameter names as the overridden ones (RQ4). Consequently, in most cases resolving method invocation with static analysis is sufficiently accurate for a name-based analysis.

Table 1: Reasons for dissimilarity between arguments and parameters.

Reasons for Dissimilarity	All Java	Java Primitive	Java API	C
Single character parameter names	24%	29%	32%	10%
Short parameter names (at most 3 characters)	43%	48%	54%	42%
Collection-related parameter names	12%	15%	16%	3%

3.5 Reasons for Dissimilarity

To answer the question why some arguments are dissimilar to their corresponding parameters, we analyze arguments that are completely dissimilar with their corresponding parameter names.

Among 754,710 of the Java arguments, 334,689 (44%) of them are completely different from their corresponding parameters (the similarity is zero). We randomly sampled 200 of them and manually inspected them. The manual analysis leads to the following two findings:

First, short parameter names are one of the reasons for the dissimilarity between arguments and parameters. 27% of the analyzed arguments are assigned to parameters named with a single character, e.g., `t` and `q`, and 46% are assigned to parameters named with at most 3 characters.

Second, collection-related parameter names lead to dissimilarity between arguments an parameters as well. 12% of the analyzed arguments are assigned to parameters named as `index`, `item`, `key`, or `value`. Such parameter names are popular in methods that manipulate data collections. Although such parameter names are meaningful, the corresponding arguments are usually dissimilar to them because the arguments are often concrete values or indexes. For example, invocation of method `Map.put(key, value)` on an array often results in a method call like `add(id, name)`.

Third, abbreviations, e.g., "ctx" and "p", result in 12%(=24/200) of the dissimilar argument-parameter pairs. However, arguments and parameters may employ different abbreviations (or the full term itself) of the same term, which results in lexical mismatch between argument and parameter names. It is challenging to predict expansions for abbreviations because an abbreviation may be expanded into different words. For example, "p" may stand for "project" or "pointer". As a result, it is challenging to measure the lexical similarity between names containing abbreviations.

To investigate whether the analysis results on the 200 sample arguments could be generalized, we validate the results on the entire population of arguments that are dissimilar to their corresponding parameters. The validation results suggest that the conclusions drawn with manual analysis on 200 sample arguments hold for the entire population of arguments. Results suggest that:

- 80,256 (24%) out of the 334,689 arguments are assigned to parameters named with a single character. 144,314 (43%) out of the 334,689 arguments are assigned to parameters named with no more than 3 characters.
- 41,484 (12%) out of the 334,689 arguments are assigned to `index`, `item`, `key`, or `value`.

We perform a similar analysis to other arguments, e.g., Java primitive type arguments, Java API arguments, and C arguments. The results are presented in Table 1. From this table we observe that short parameter names and collection-related parameter names are the major reasons for the dissimilarity regardless of the types of such arguments. However, we also observe that the popularity of such names varies among different kinds of arguments. For example, 42% of the dissimilar pairs of C arguments and parameters are caused by short parameter names. In contrast, the ratio increases significantly to 56% for Java API arguments.

We conclude from these results that short parameter names, collection-related parameter names, and abbreviations are major reasons for the dissimilarity between argument and parameter names (RQ5). These findings suggest that applications based on name similarities may improve their effectiveness by distinguishing (and ignoring) such names.

3.6 Filtering Low Similarity Parameters

This section investigates the question whether one can build a set of low-similarity parameters from a corpus of sample applications, and whether such low-similarity parameters are likely to be assigned to dissimilar arguments in other applications as well.

Given a corpus of sample applications, we identify low-similarity parameters according to the following three steps. First, we cluster all argument names in the sample applications according to their corresponding parameter names. Each cluster is associated with a unique parameter name. Second, for each cluster, we calculate the average similarity s between the parameter and each argument in this cluster. 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 parameter names.

Table 2 illustrates how arguments associated with low-similarity parameters change the distribution of similarity between Java arguments and parameters. From the table, we observe that Java arguments that are less similar to their parameters are more likely to be filtered out. For example, the filtration removes 81% of the Java arguments whose similarity with their parameters is zero, whereas it removes only 20% of the Java arguments whose similarity with parameters is not zero.

The distribution of similarity between parameters and arguments that are not associated with low-similarity parameters is presented in Fig. 12. By comparing Fig. 12 against Fig. 1, we observe that the distribution of lexical similarity between arguments and parameters has been reshaped dramatically by

Table 2: 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)	334,689	271,945	81%
[0.1, 0.2)	3	2	67%
[0.2, 0.3)	534	416	78%
[0.3, 0.4)	1,442	907	63%
[0.4, 0.5)	5,649	3,085	55%
[0.5, 0.6)	29,922	10,592	35%
[0.6, 0.7)	53,088	12,683	24%
[0.7, 0.8)	160	6	4%
[0.8, 0.9)	6,735	465	7%
[0.9, 1.0)	14	1	7%
1	322,474	55,648	17%
Total	754,710	355,750	47%

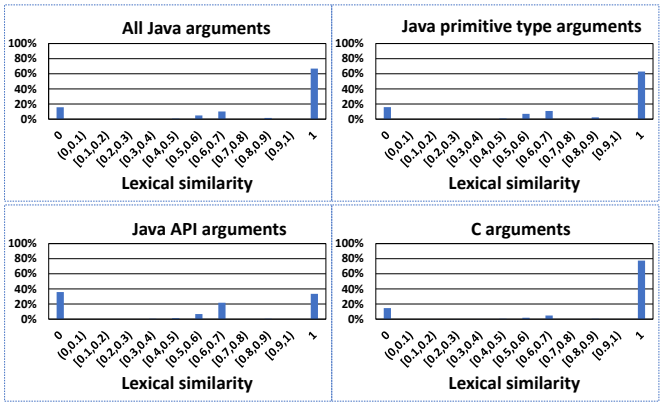

Fig. 12: Distribution of similarity after filtering low-similarity parameters.

filtering low-similarity parameters. For example, the ratio of Java arguments that are highly dissimilar (the similarity is zero) to their parameters decreases from 44% to 16%.

Filtering out low similarity parameters is much better than simply removing API methods. Removing API methods may remove a large number of low similarity parameters because API parameters are often dissimilar with arguments as shown in Session 3.1. We apply these two approaches (i.e., filtering out low similarity parameter names, and simply removing API methods) independently, and results suggest that the latter is less effective: Still up to 31% of the arguments are completely different from their corresponding parameters after API methods are removed. In contrast, the filtration reduces the number dramatically to 16%. Only 55% of the arguments are identical to

their corresponding parameters even if API methods are removed. However, after filtering out low-similarity parameters, we have 67% arguments that are identical to their corresponding parameters.

We conclude from these results that building a list of low-similarity parameters based on a corpus of sample applications is an effective means to exclude dissimilar pairs of arguments and parameters (RQ6). This finding enables approaches that exploit name similarities to improve the precision by ignoring low-similarity parameters.

3.7 Picking Among Alternative Arguments

This section investigates the question how often the current argument chosen by the developer is more similar to the corresponding parameter than alternative arguments, we compare each argument against its potential alternatives. Results suggest that after filtering out arguments associated with low-similarity parameter names, the majority (66%) of the arguments are more similar to their corresponding parameters than any of their alternatives.

Definition 1 An *alternative argument* of argument arg is a potential alternative that does not introduce new syntactical or type errors when replacing arg.

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 (i.e., private fields), or their types may be incompatible with the parameter's type. Therefore we exclude such invalid potential alternatives from the comparison from *alternative arguments*.

Among the alternative arguments (alt_args), we refer to the argument that has the greatest lexical similarity with the corresponding parameter par as *the most similar potential argument*:

Definition 2 An alternative argument $m\_alt \in alt\_args$ is *the most similar potential argument* if for any alternative argument $any\_alt \in alt\_args$, the following expression holds: $tSim(m\_alt, par) \geq tSim(any\_alt, par)$

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 to the names of its alternative arguments.

The results are presented in Table 3. From this table, we observe that 33% (249,438) of the 754,710 analyzed Java arguments have at least one alternative argument. Of such arguments that have alternatives, 43% (=106,191/249,438) are more similar to the parameter than any alternatives, 56% have an alternative that is similar to the same degree, and only 11% have an alternative that is more similar to the parameter. For those arguments whose similarity with the corresponding parameters is greater than 0.5, up to 89% of the them are more similar to the parameter than any alternative.

Table 3: Picking among alternative arguments.

		Java	Primitive type	API	C
Before filtering	No. Arguments (n_1)	754,710	269,485	199,441	345,602
	No. Arguments with alternatives (n_2)	249,438	163,940	61,797	93,928
	No. Arguments more similar to the parameter than alternatives (n_3)	106,191	79,320	9,579	55,629
	n_2/n_1	33%	61%	31%	27%
	n_3/n_2	43%	48%	16%	59%
After filtering	No. Arguments (n_4)	398,960	148,376	23,264	266,113
	No. Arguments with alternatives (n_5)	128,537	89,726	9,847	62,094
	No. Arguments more similar to the parameter than alternatives (n_6)	85,254	64,930	4,878	44,949
	n_5/n_4	32%	60%	42%	23%
	n_6/n_5	66%	72%	50%	72%

We also analyze the impact of filtering arguments associated with low-similarity parameters. Results suggest that most arguments that are not associated with low-similarity parameters are more similar to their corresponding parameters than any other alternative arguments. After the filtration, we keep 398,960 arguments, and 128,537 (32%) of them have at least one alternative. 66% of such arguments are more similar to the corresponding parameters than any of their alternatives. Compared to that (43%) before filtration, the ratio (66%) has increased by 53%=(66%-43%)/43%.

The conclusions drawn on Java arguments hold for primitive, API, and C arguments as well. Most of the arguments that are not associated with low-similarity parameters are more similar to their corresponding parameters than any other alternative arguments. 61% of the primitive arguments, 31% of the API arguments, and 27% of C arguments have at least one alternative argument. After filtering arguments associated with low-similarity parameters, 72% of the primitive arguments, 50% of the API arguments and 72% of the C arguments are more similar to the parameters than any of their alternatives.

We conclude from these results that name-based approaches that try to infer potential arguments, such as code completion or anomaly detection, may have a high chance to make accurate suggestions if they filter out arguments with a list of low-similarity parameter names (RQ7).

3.8 Influence of the Similarity Metrics

To assess to what degree the major conclusions drawn in previous sections hold when different lexical similarity metrics are employed, we repeat the analysis with two typical lexical similarity metrics, i.e., *Levenshtein distance-based* and *JaroWinkler-based* similarity metrics. It should be noted that it is not to validate which metric is better than other metrics. The only purpose of the section is to validate whether the conclusions drawn with name-based argument analyses work well with diverse types of similarity metrics. If there is no essential difference in the evaluation results when different similarity metrics are employed, it hints that the conclusions drawn in previous sections do not depend on specific similarity metrics, which generalizes the conclusions. If

different similarity metrics lead to essentially different properties of argument and parameter names, the evaluation results may provide hints for the selection of similarity metrics in name-based argument analyses tasks.

3.8.1 Alternative Metrics

For the Levenshtein distance-based similarity metrics, we compute the similarity of two names as follows. First, we calculate the Levenshtein distance $LevDistance(arg, par)$ between the argument name arg and the parameter name par. Levenshtein distance is a way to quantify how dissimilar two strings are by counting the minimum number of character operations (e.g., insertion, deletion, substitution) required to transform one string into the other. For example, $LevDistance(\text{“}name\text{”}, \text{“}getName\text{”}) = 3$. Second, we compute the Levenshtein distance-based similarity between the argument name arg and the parameter name par as follows:

$$LevSim(arg, par) = \frac{maxLength(arg, par) - LevDistance(arg, par)}{maxLength(arg, par)} \tag{8}$$

$maxLength(arg, par)$ stands for the max length of arg and par. $LevSim$ gives the fraction of the longer name that matches the shorter one and does not have to be edited. For example, $LevSim(\text{“}name\text{”}, \text{“}getName\text{”}) = \frac{7-3}{7} = 57\%$.

For the JaroWinkler-based similarity metrics (called $jwSim$ for short), we compute the similarity $jwSim$ between the argument name arg and the parameter name par, which is based on the number and order of the common characters between the two names, as follows:

$$jwSim(arg, par) = jaroSim(arg, par) + len * p(1 - jaroSim(arg, par)) \tag{9}$$

where p is a constant scaling factor for how much the score is adjusted upwards for having common prefixes, len is the length of common prefix at the start of the string, and $jaroSim$ is computed as follows:

$$jaroSim(arg, par) = \frac{1}{3}(\frac{m}{|arg|} + \frac{m}{|par|} + \frac{m-r}{m})$$

where m is the number of matching characters, and r is the number of matching (but different sequence order) characters.

3.8.2 Results

For space limitation, the detailed results, including figures, are presented on an online appendix (app 2017-9-23). Here we introduce the major findings.

First, the major conclusions drawn in preceding paragraphs hold when alternative metrics are employed:

- Most arguments are either extremely similar or extremely dissimilar with corresponding parameters. For convenience, we call such arguments *extreme arguments*. They account for 87%, 59%, and 70% of the analyzed arguments when term-based similarity, Levenshtein distance-based similarity, and JaroWinkler-based similarity are employed, respectively.
- The similarity increases when the length of parameter names increases, whereas it decreases when the length of argument names increases. The trends keep unchanged while different similarity metrics are employed.
- Short parameter names and collection-related parameter names are the major reasons for the dissimilarity. They lead to 56%, 76%, and 69% of the dissimilar argument-parameter pairs when term-based similarity, Levenshtein distance-based similarity, and JaroWinkler-based similarity are employed, respectively.
- Dissimilar argument-parameter pairs can be filtered out by building a filter list of low similarity parameter names. 81%, 81%, and 75% of the dissimilarity argument-parameter pairs are filtered out when term-based similarity, Levenshtein distance-based similarity, and JaroWinkler-based similarity are employed, respectively.
- Most arguments that are not associated with low-similarity parameters are more similar to their corresponding parameters than any other alternatives. The ratio is up to 66%, 78%, and 75% when term-based similarity, Levenshtein distance-based similarity, and JaroWinkler-based similarity are employed, respectively.

Besides these commonalities, we also observe some differences. A significant difference is that Levenshtein distance-based similarity metrics greatly reduce the number of arguments whose similarity with their corresponding parameters is zero (completely different). It is nature in that Levenshtein distance-based similarity metrics take two terms as completely different only if they share no common characters at all. In contrast, term-based similarity metrics take them as completely different whenever they are not identical (but may share some common characters).

We conclude that the major conclusions drawn in Sections 3.1 - 3.7 hold as well when term-based similarity metrics are replaced with Levenshtein distance-based or JaroWinkler-based similarity metrics (RQ10).

4 Applications

The empirical study presented in the preceding sections leads to a number of interesting and valuable findings concerning arguments and parameters. Such findings may facilitate argument-related software engineering tasks. In this section, we explore two such analyses.

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, possible 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 identify 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 2). 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.

The approach 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, it identifies and reports suspicious arguments immediately when the developer introduces them and suggests to the developer an alternative arguments as a quick-fix. Second, to check all arguments in a project at once. In this scenario, it checks the whole application and reports all suspicious arguments at once, along with the source code location of each problem and suggestions for alternative arguments.

4.1.2 Calibration

The approach depends on a threshold β 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). *Openbravo POS* is a sale application supporting touch screens. *Domination* is an application to create games for Android. *Dom4j* is an open-source Java library for working with XML, XPath and XSLT. 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. Every reported warning is manually checked by a team that is composed of three engineers. A reported warning is considered a *true positive* if it points to a valid renaming opportunity or to an incorrect argument. In all cases (41 warnings reported in total), the engineers reach an agreement after discussion. Based on this classification, we compute the precision of the anomaly detection as follows:

$$Precision = \frac{\text{Number of true positives}}{\text{Number of reported warnings}} \tag{10}$$

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 connects such discrete points with a dashed line, and observe that the precision increases while β increases from 0.4 to 0.667 (=2/3), and it decreases slightly after this point. Based on these results, we use $\beta = 0.667$ in the remaining experiments.

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 (Fluri et al 2007) 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 is designed to ensure 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: upload-rate 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. For example, Figure 13 lists two example bugs. We then apply the analysis to the buggy versions of the 11 applications. To decide whether a reported anomaly is an incorrect argument, a renaming opportunity, or a false positive, three engineers manually check all warnings reported by the approach and discuss each warning until a consensus is reached.

We apply the anomaly detection to the 11 subject applications with known incorrect arguments (Table 4). 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 as introduced in the preceding paragraph. The results may suggest that incorrect arguments are more popular than ChangeDistiller-based checking suggests.

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);
```

Fig. 13: Examples of incorrect arguments detected by the anomaly detection.

Table 4: Evaluation 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.*

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:

• *Abbreviation* (42/127=33%). For example, the approach warns about an argument `c` whose corresponding parameter is `country`, and about a name `ds` that stands for "data source".

• *Incomplete descriptions that misses nouns* (30/127=24%). Identifier names often consist of a noun combined with some adjunct. The approach warn-

s about several argument names where the noun is missing, such as `missed`, which should be renamed into `missedMembers` (corresponding parameter: `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. It may suggest that the quality of parameter names is often 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 (Lawrie et al 2006), 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 to swap 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 fixed by more sophisticated processing of names, such as Butler et al.'s method for tokenizing identifier names (Butler et al 2011) or techniques borrowed from the natural language processing community.

An earlier version of this paper has inspired Google to implement a name-based bug detection technique, similar to the anomaly detection discussed here (Rice et al 2017). Their work adds several heuristics to remove false positives.

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

Table 5: Results of Argument Recommendation.

Ten-fold	Recommended	Accepted	Rejected	Precision	Recall
1#	62,847	46,921	15,926	74.66%	48.24%
2#	60,207	40,091	20,116	66.59%	41.22%
3#	56,605	42,713	13,892	75.46%	43.91%
4#	64,501	47,288	17,213	73.31%	48.62%
5#	64,476	46,065	18,411	71.45%	47.36%
6#	59,856	39,623	20,233	66.20%	40.74%
7#	62,057	45,642	16,415	73.55%	46.92%
8#	60,159	44,254	15,905	73.56%	45.50%
9#	59,312	46,355	12,957	78.15%	47.66%
10#	60,739	46,986	13,753	77.36%	48.31%
Average	61,076	445,94	16,482	73.03%	45.85%

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}) passing any of which as the actual argument will not introduce syntactical errors. It should be noted that complex expressions, type conversion, or literals, e.g., `2*height` and `99`, are not included because considering such complex expressions, type conversion, or literals would make the search space for potential arguments extremely large. Third, it computes the similarity between the parameter name and the names of the collected potential arguments. Finally, if one element from S_{pot} whose similarity is the maximum one, the approach recommends this element.

4.2.2 Evaluation

We search for open-source C applications with most stars from Github (git 2016-8-2) that have at least 5 releases and can be imported into Eclipse. We select the top 85 resulting applications satisfying such constraints, and extract 972,660 non-API arguments (and their context) from such applications. With such arguments, we carry out a ten-fold cross-validation. The arguments are randomly partitioned into ten equally sized groups notated as G_i ($i = 1 \cdots 10$). For the ith cross-validation, we consider all arguments except for those in G_i as the corpus of training data. For the ith cross-validation, the evaluation process is as follows:

1. First, we extract training data set $traData_i$ that is the union of all groups but G_i: $traData_i = \bigcup_{j \in [1,10] \wedge j \neq i} G_j$.
2. Second, we compute a list (l_i) of low similarity parameter names based on $traData_i$.
3. Third, for each argument in G_i, we make recommendation based on the approach presented in Section 4.2.1, and compare the recommended arguments against the chosen (correct) argument.
4. Finally, we compute the precision and recall of the recommendation.

The evaluation takes the assumption that arguments in the subject applications are correct. Consequently, a recommendation is accepted (correct) if and only if the recommended argument is identical to the current one appearing in the source code.

The evaluation results are presented in Table 5. From the table, we observe that the approach is accurate in recommending arguments. Among the 610,759 recommended arguments , 73.03%(=445,938/610,759) are accepted (correct). Its precision varies from 66.2% to 78.15%, with an average of 73.03%. Its recall varies slightly from 40.74% to 48.62%, with an average of 45.85%.

One of the reasons why the approach fails in some cases is that there are quite a few unsupported arguments in the subject applications. As introduced in Section 4.2.1, complex expressions, literals and type conversions are not supported by the approach. Such unsupported arguments account for 14.86% (=144,539/972,660) of the arguments in the subject applications. They have significant negative impact on the effectiveness (both precision and recall) of the proposed approach because it can never succeed when the correct argument is one of the unsupported arguments.

5 Threats and Limitations

5.1 Threats

A threat to external validity is that the conclusions in this study are drawn from only a small set of applications. As a result, the conclusions drawn on these applications may not hold on other applications. To reduce the threat, we select 125 most popular open-source applications from the well-known open-source community *SourceForge*. 60 of these applications have been analyzed in our previous conference paper. However, the adware scandal[6] of *SourceForge* suggests that source code from *SourceForge* may have been modified in some way, which is a threat to external validity. To minimize the threat, the analysis includes 30 applications from *GitHub* as well as. Analyzing open-source applications only could be a threat to external validity. Consequently, to reduce the threat, we include 2 closed-source applications for the analysis. As

[6] https://www.infoworld.com/article/2929732/open-source-software/sourceforge-commits-reputational-suicide.html

a result, the analysis includes 125 open-source applications from *SourceForge*, 30 open-source applications from *GitHub* , and 2 closed-source applications.

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

The third threat to external validity is that all of the applications are implemented in Java and C. Conclusions drawn on Java and C applications might not hold for other programming languages. Java and C are selected because Java is a popular object-oriented programming language, whereas C is a popular procedure-oriented programming language. There are essential differences between Java and C arguments. First, field accesses are frequently used as arguments in Java whereas it is not allowed in procedure-oriented languages (e.g., C) because there are no fields at all. Second, pointers could be used as arguments in C, but not in Java. Third, cascaded method invocations, e.g., *obj.getInst().getObj()*, are frequently used as Java arguments, but they are rarely employed in C. Validating on such diverse data set may help to increase the generalizability of the results.

A threat to construct validity is that the lexical similarity between argument and parameter names may not be properly computed. Up to date there is no golden standard for measurement of lexical similarity between two identifiers. To reduce the threat, we employ multiple popular similarity metrics, i.e., term-based similarity metrics, Levenshtein distance-based similarity metrics, and JaroWinkler-based similarity metrics.

Another threat to construct validity is that for some non-variable arguments, e.g., method invocation, the way to extract their identifier names may be misleading. For example, we simply take the method name of the invoked method as the argument name. However, such names often result in mismatch between argument and parameter names because method names and parameter names often have different syntactic structures. In the future, it would be interesting to evaluate other approaches, e.g., taking the name of the returned variable as the argument name.

Our evaluation of the effectiveness of a name-based anomaly detection is subject to two internal 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.

5.2 Limitations

A limitation of the lexical similarity metrics employed in this paper is that it cannot properly recognize the lexical similarity between some special terms, e.g., abbreviations vs. their expansions, and singular vs. plural. It is likely that such terms may appear in identifiers, although they may not be popular. For example, among the 754,710 argument-parameter pairs analyzed in the experience study, 545 of them have encountered the mismatch of singular and plural, i.e. the argument name contains the single form of a term whereas the parameter name contains the plural form of the same term (or the other way around). The term-based similarity metrics takes the singular and plural (of the same term) as different terms because they are not identical. However, edit distance-based similarity metrics may recognize their similarity because they often contain a large number of common characters. For example, the distance between "student" and "students" is up to $(8-1)/8 = 7/8 = 0.875$. In the future, it would be valuable to employ advanced techniques to identify abbreviations and plural (Hill et al 2008; Sun et al 2014), which may further improve the similarity measurement. However, the identification, especially the identification of abbreviations, is challenging. A survey conducted by Beniamini et al. (Beniamini et al 2017) suggests that certain commonly used short and even single-letter names are strongly associated with certain types and meanings. For example, "i" is often used in loop statement as a counter, can thus it may be taken as an abbreviation. It is difficult to automatically identify the meaning (or expansion) of such extremely short identifier names.

Another limitation is that we cannot recognize inclusion relation between concepts represented by terms. It is common that the generic parameter name of an API method presents a generic and big concept (e.g., component) whereas its corresponding argument name represents a concrete and small concept (e.g., button). Our lexical similarity metrics cannot reflect such kind of conceptual relationship: the latter is a subset of the former. In future, it would be interesting to employ more advanced technologies, e.g., ontology analysis, to enhance the proposed approach.

6 Disscussion

The empirical study presented in the preceding sections leads to a number of interesting and valuable findings concerning arguments and parameters. Such findings may facilitate argument-related software engineering tasks, e.g, incorrect argument detection and argument recommendation. Two concrete applications of the findings have been presented in the conference paper (Liu et al 2016). We briefly discuss these applications in the following and refer to (Liu et al 2016) for details, because the focus of this paper is the empirical study presented in preceding sections.

One of the findings suggests that correct arguments are often more similar to their corresponding parameters than alternatives. Consequently, if an

argument is significantly less similar to the parameter than its alternatives, the reason may be that the argument is incorrect. Another finding suggests that many dissimilar but nevertheless correct arguments could be filtered out with a list of low-similarity parameter names built on sample applications. Based on these two findings, a name-based anomaly detection approach has been proposed (Liu et al 2016). The approach effectively detected 6 of 14 known incorrect arguments across 11 open-source applications, as well as 3 additional, previously unknown incorrect arguments. In addition to the incorrect arguments, the approach reported 127 renaming opportunities and 33 false positives.

Another application of our study is name-based argument recommendation (Liu et al 2016). 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. The approach was evaluated on four open-source applications, and it recommended 1,588 arguments with a precision of 83%.

Beyond these two applications, we envision our study to encourage future research on name-based program analysis. The findings reported here may guide that research by providing insights on the properties of argument names and parameter names in real-world code written in Java and C.

7 Related Work

7.1 Importance of Identifier Names

Identifiers play an important role in source code comprehensibility and maintainability (Eshkevari et al 2011). Identifier style affects the speed and accuracy of comprehending programs, and that beginners benefit from the popular camel case naming style (Binkley et al 2013). Identifiers complying with naming conventions improve program understandability (Salviulo and Scanniello 2014). Butler et al. (Butler et al 2010) found that flawed identifiers in Java classes were associated with source code found to be of low quality by static analysis. They use a set of empirically evaluated identifier naming guidelines to measure the identifier quality and employ the cyclomatic complexity metrics and the maintainability index to evaluate the quality of source code. Caprile et al. (Caprile and Tonella 1999, 2000) believe that identifiers convey relevant information about the role and properties of the objects they are intended to label. Identifiers are the starting point for the program understanding activities, especially when high level views, i.e., the call graph, are available. Lawrie et al. (Lawrie et al 2006) believe that identifier names, as well as comments, are the main sources of domain information from source code. Because many developers do not write comments, identifier names are in fact the most critical source for program comprehension. However, most of the data in identifier names and comments are unstructured data. Researchers often employ NLP techniques, e.g., tokenization, splitting, and stemming, to extract the

information embedded in data. Sun et al. (Sun et al 2014) found that data preprocessing techniques employed affect the comprehension of identifier names. Arnaoudova et al. (Arnaoudova et al 2014) suggest that meaningful identifier names can be exploited to automatically obtain the initial idea of the role of each entity and give a concise description of the program behavior and functionality. Ohba et al. (Ohba and Gondow 2005) propose a novel technique, *ckTF/IDF*, to mine concept keywords from identifiers in large software projects. Liblit et al. (Liblit et al 2006) find that developers choose identifier names in regular and semantic ways, which conveys the cognitive and linguistic background of developers. Lexical and morphological convention of the identifier reflect the role of it.

7.2 Quality of Identifier Names

Because of the importance of identifier names, researchers have proposed a number of approaches to improve the quality of poor names. Such approaches should identify poor names first, and replace them with better names, which is well-known as rename refactoring. To identify bad names in general, we should analyze the meaning of identifier names (composed of English terms) and the meaning of the named software entities. The analysis involves complex natural language processing (NLP) as well as complex sematic analysis of source code. As a result, it's challenging to decide which names are poor.

Although it is challenging in general to decide whether a given identifier name is appropriate, approaches have been proposed to identify special categories of bad names. The first category of such approaches to identifying bad names checks identifier names against predefined rules. For example, Abebe et al. (Abebe et al 2009) introduce *lexicon bad smells* that indicate potential lexicon construction problems, Caprile and Tonella (Caprile and Tonella 2000) propose an approach to standardize program identifier names, Corbo et al. (Corbo et al 2007) propose an approach to identify identifiers that violate naming conventions by analyzing the structure of identifiers from existing source code. Allamanis et al. (Allamanis et al 2014) present a framework, *NATURALIZE*, to learn the code conventions and suggest consistent identifier names. The second category of such approaches to identifying bad names searches for inconsistent naming. Host et al. present a series of approaches to identify inconsistencies between the method names and implementations based on syntactic analysis of the name and semantic analysis of the implementation (Høst and Østvold 2007, 2009b,a; Karlsen et al 2012). Deissenboeck and Pizka (Deissenboeck and Pizka 2006) propose a model-based approach for identifying inconsistent naming. Arnaoudova et al. propose a Linguistic Antipatterns(LAs) based approach to detect inconsistency between method names and method signatures (Arnaoudova et al 2013). The approach can also detect inconsistency between method names and method comments.

Eshkevari et al. (Eshkevari et al 2011) study the types of renamings, the frequencies of renamings, and the locations of renamings. They find that syn-

onym, hyponym, hypernym, and antonym could be used to improve code quality in renaming recommendation system.

The findings in this study may help to identify bad names as well. As suggested by Section 3.6, arguments that are not associated with low-similarity parameters are often highly similar to their corresponding parameters. Consequently, we can detect bad argument name or parameter name based on the lexical similarity between arguments and parameters.

7.3 Approaches Based On Name Similarity

Pradel and Gross (Pradel and Gross 2011, 2013) present an approach to identify problems related to the order of equally typed arguments based on the similarities between names used at different positions. If the names of arguments deviate from typically used names for the given method and a different argument order match the common pattern used at other call sites significantly better, they report a warning and suggest to change the order of related arguments. They also present an approach to detect incorrect parameter names when many call sites agree on a naming scheme for arguments but the parameter names do not reflect the naming scheme.

Cheng et al. (Cheng et al 2016) present an approach to detect cross-platform clones based on the identifier similarity. They hypothesize that developers use common identifiers when they implement similar functionalities on different platforms. They use the similarity of token distributions, KL-Divergence (Kullback and Leibler 1951), to represent the similarity of source code. They find that the smaller KL-Divergence is, the more accurate the approach is.

Allamanis et al. (Allamanis et al 2015) present a neural probabilistic language model to recommend method and class names by learning the semantic similarity between names. First, they convert each identifier into a continuous vector in a high dimensional space, where identifiers occurs in similar contexts are with similar vector. Second, they suggest the one, who is most similar to the method body in the high dimensional space, as the name for the method.

Sridhara et al. (Sridhara et al 2011) present a technique to automatically generate comments for parameters of Java methods, based on static analysis of source code only. They take Java method as input, generate a summary for the method using the summary comment generator (Sridhara et al 2010), analyze the main role of each parameter in the method based on the syntax, semantic, and linguistic information embedded in the identifier names, and generate descriptive comments based on advanced text generation techniques. Evaluation by experienced developers suggests that the generated comments for parameters are accurate and descriptive.

The findings in this study suggest several applications that exploit the lexical similarity between arguments and parameters as well. As suggested in Section 3.7, after filtering arguments associated with low-similarity parameters, name-based approaches may have a high chance to make accurate suggestions

for potential arguments. Consequently, we can detect bad names when the current argument is significantly less similar to the parameter than an alternative and recommend an argument from a set of potential arguments so that the recommended argument is the most similar to the corresponding parameter.

A number of approaches have been proposed to calculate name similarity for identifiers. Cohen et al. (Cohen et al 2003) compare different string metrics for matching names and records, including edit-distance metrics, fast heuristic string comparators, token-based distance metrics, and hybrid metrics. They propose a hybrid similarity metric.

The first similarity metrics Cohen et al. investigated (Cohen et al 2003) are edit distance similarity metrics. Edit distance is widely used to quantify how dissimilar two strings are by counting the minimum number of character-based operations required to transform one string into the other (Monge and Elkan 2001). The more similar the two strings are, the smaller the edit distance is.

The second similarity metrics Cohen et al. investigated (Cohen et al 2003) are Jaro-Winkler similarity metrics, proposed by Winkler (Winkler 1999) for comparison of short strings. The comparison is based on the number and order of the common characters between two strings.

The third similarity metrics Cohen et al. investigated (Cohen et al 2003) are Jaccard similarity metrics. The more similar the two strings are, the larger the Jaccard similarity is.

The fourth similarity metrics Cohen et al. investigateds (Cohen et al 2003) are Cosine similarity metrics. Cosine (or TFIDF) similarity, one of the two wide-used token-based similarity metrics, is a measure of similarity between two non-zero vectors. It measures the cosine of the angle between the two vectors. TFIDF is used in information retrieval to compare whole documents with large vocabularies. However, in our context where most identifier names are composed of a single term, TFIDF may not be useful.

In this study, we try the term-based similarity metrics, the Levenshtein distance-based similarity metrics (based on characters), and the JaroWinkler-based similarity metrics (based on terms) to measure the similarity between arguments and parameters. In future, it would be interesting to integrate more similarity metrics to measure the similarity between arguments and parameters.

Arnaoudova et al. (Arnaoudova et al 2014) exploit ontological databases, i.e., WordNet (Miller 1995), to facilitate comparison between identifier names. They build the taxonomy based on grounded-theory in dimensions that apply to source code identifiers and the terms that compose the taxonomy. In future, it would be interesting to improve the computation of lexical similarity between identifier names (introduced in Section 2.2) by exploiting such ontological databases.

Fritz et al. (Fritz et al 2014) introduce a degree-of-knowledge (DOK) model that computes automatically who knows the particular parts of code and who needs to know the change of code, based on the authorship and interaction data of the code. The DOK model performs well on expert finding, knowledge transfer, and identifying changes of interest.

8 Conclusions and Future Work

This paper presents an empirical study of the lexical similarity between argument and parameter names. The following findings are revealed by this study. First, argument names are often identical to or completely different from their corresponding parameter names. Second, dissimilar pairs of arguments and parameters can be filtered out based on a set of low-similarity parameters inferred from a set of sample programs. Third, many arguments are more similar to their corresponding parameters than alternative arguments that are available in the scope of the method calls. These observations hold across different languages (C and Java) and different type of arguments/parameters (primitive versus non-primitive, API versus non-API).

Measures of similarity between identifiers serves as the basis of the research. We have tried term-based, Levenshtein distance-based, and JaroWinkler-based similarity metrics for similarity, but we have not found major changes in the overall results of the study. In future, it would be interesting to try additional alternative metrics for similarity.

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.

References

(2016-5-8) http://sourceforge.net/
(2016-8-2) http://github.com/
(2017) p-value. Encyclopedia of Computational Neuroscience p 2548
(2017-9-23) https://github.com/D12126977/appendix
(2017-9-23) https://github.com/D12126977/dataset
Abebe S, Haiduc S, Tonella P, Marcus A (2009) Lexicon bad smells in software. In: 16th Working Conference on Reverse Engineering (WCRE '09), pp 95–99, DOI 10.1109/WCRE.2009.26
Allamanis M, Barr ET, Bird C, Sutton C (2014) Learning natural coding conventions. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, FSE 2014, pp 281–293, DOI 10.1145/2635868.2635883, URL http://doi.acm.org/10.1145/2635868.2635883
Allamanis M, Barr ET, Bird C, Sutton C (2015) Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp 38–49, DOI 10.1145/2786805.2786849, URL http://doi.acm.org/10.1145/2786805.2786849
Arnaoudova V, Di Penta M, Antoniol G, Guéhéneuc YG (2013) A new family of software anti-patterns: Linguistic anti-patterns. In: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '13, pp 187–196, DOI 10.1109/CSMR.2013.28, URL http://dx.doi.org/10.1109/CSMR.2013.28

Arnaoudova V, Eshkevari LM, Di Penta M, Oliveto R, Antoniol G, Guéhéneuc YG (2014) REPENT: Analyzing the nature of identifier renamings. IEEE Transactions on Software Engineering 40(5):502–532, DOI 10.1109/TSE.2014.2312942

Beniamini G, Gingichashvili S, Orbach AK, Feitelson DG (2017) Meaningful identifier names: The case of single-letter variables. In: Ieee/acm International Conference on Program Comprehension, pp 45–54

Binkley, Dave, Lawrie, Morrell, Christopher, Davis, Marcia, Maletic, Jonathan I (2013) The impact of identifier style on effort and comprehension. Empirical Software Engineering 18(2):219–276

Butler S, Wermelinger M, Yu Y, Sharp H (2010) Exploring the influence of identifier names on code quality: An empirical study. In: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, pp 156–165, DOI 10.1109/CSMR.2010.27

Butler S, Wermelinger M, Yu Y, Sharp H (2011) Improving the tokenisation of identifier names. In: Mezini M (ed) European Conference on Object-Oriented Programming (ECOOP 2011), Lecture Notes in Computer Science, vol 6813, Springer Berlin Heidelberg, pp 130–154

Caprile B, Tonella P (1999) Nomen est omen: Analyzing the language of function identifiers. In: Reverse Engineering, 1999. Proceedings. Sixth Working Conference on, IEEE, pp 112–122

Caprile B, Tonella P (2000) Restructuring program identifier names. In: Software Maintenance, Proceedings. International Conference on, pp 97–107, DOI 10.1109/ICSM.2000.883022

Cheng X, Jiang L, Zhong H, Yu H, Zhao J (2016) On the feasibility of detecting cross-platform code clones via identifier similarity. In: International Workshop on Software Mining, pp 39–42

Cohen WW, Ravikumar PD, Fienberg SE (2003) A comparison of string distance metrics for name-matching tasks. In: Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), pp 73–78

Corbo F, del Grosso C, Di Penta M (2007) Smart formatter: Learning coding style from existing source code. In: Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, pp 525–526, DOI 10.1109/ICSM.2007.4362682

Deissenboeck F, Pizka M (2006) Concise and consistent naming. Software Quality Journal 14(3):261–282, DOI 10.1007/s11219-006-9219-1, URL `http://dx.doi.org/10.1007/s11219-006-9219-1`

Eshkevari LM, Arnaoudova V, Penta MD, Oliveto R, Guéhéneuc YG, Antoniol G (2011) An exploratory study of identifier renamings. In: International Working Conference on Mining Software Repositories, MSR, pp 33–42

Fluri B, Wrsch M, Pinzger M, Gall HC (2007) Change distilling: Tree differencing for fine-grained source code change extraction. pp 725–743

Fritz T, Murphy GC, Murphy-Hill E, Ou J, Hill E (2014) Degree-of-knowledge: Modeling a developer's knowledge of code. Acm Transactions on Software Engineering & Methodology 23(2):14

Hill E, Fry ZP, Boyd H, Sridhara G, Novikova Y, Pollock L, Vijay-Shanker K (2008) AMAP:automatically mining abbreviation expansions in programs to enhance software maintenance tools. In: International Working Conference on Mining Software Repositories, pp 79–88

Høst EW, Østvold (2007) The Programmer's Lexicon, Volume I: The Verbs. In: SCAM 2007. Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, 2007., pp 193–202, DOI 10.1109/SCAM.2007.18

Høst EW, Østvold BM (2009a) Debugging method names. In: European Conference on ECOOP 2009 — Object-Oriented Programming, pp 294–317

Høst EW, Østvold BM (2009b) The Java programmers phrase book. In: Software Language Engineering, pp 322–341

Karlsen EK, Høst EW, Østvold BM (2012) Finding and fixing Java naming bugs with the Lancelot Eclipse plugin. In: Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, ACM, New York, NY, USA, PEPM '12, pp 35–38, DOI 10.1145/2103746.2103756, URL `http://doi.acm.org/10.1145/2103746.2103756`

Kullback S, Leibler RA (1951) On information and sufficiency. Ann Math Statist 22(1):79–86, DOI 10.1214/aoms/1177729694, URL http://dx.doi.org/10.1214/aoms/1177729694

Lawrie D, Morrell C, Feild H, Binkley D (2006) What's in a name? a study of identifiers. In: 14th International Conference on Program Comprehension, IEEE Computer Society, pp 3–12

Liblit, Ben, Begel, Andrew, Sweetser (2006) Cognitive perspectives on the role of naming in computer programs. Proc of Annual Psychology of Programming Workshop

Liu H, Liu Q, Staicu CA, Pradel M, Luo Y (2016) Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In: International Conference on Software Engineering, pp 1063–1073

Miller GA (1995) WordNet: a lexical database for English. Communications of the ACM 38(11):39–41, DOI 10.1145/219717.219748

Monge AE, Elkan CP (2001) An efficient domain-independent algorithm for detecting approximately duplicate database records. Proc of the Sigmod

Ohba M, Gondow K (2005) Toward mining "concept keywords" from identifiers in large software projects. pp 1–5

Pandita R, Xiao X, Zhong H, Xie T, Oney S, Paradkar A (2012) Inferring method specifications from natural language API descriptions. In: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '12, pp 815–825, URL http://dl.acm.org/citation.cfm?id=2337223.2337319

Pradel M, Gross TR (2011) Detecting anomalies in the order of equally-typed method arguments. In: International Symposium on Software Testing and Analysis (ISSTA), ACM, pp 232–242

Pradel M, Gross TR (2013) Name-based analysis of equally typed method arguments. IEEE Transactions on Software Engineering 39(8):1127–1143, DOI http://doi.ieeecomputersociety.org/10.1109/TSE.2013.7

Raychev V, Vechev M, Yahav E (2014) Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '14, pp 419–428, DOI 10.1145/2594291.2594321, URL http://doi.acm.org/10.1145/2594291.2594321

Raychev V, Vechev M, Krause A (2015) Predicting program properties from "big code". In: 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15), pp 111–124

Rice A, Aftandilian E, Jaspan C, Johnston E, Pradel M, Arroyo-Paredes Y (2017) Detecting argument selection defects. In: SPLASH 2017 OOPSLA

Salviulo F, Scanniello G (2014) Dealing with identifiers and comments in source code comprehension and maintenance: results from an ethnographically-informed study with students and professionals. In: International Conference on Evaluation and Assessment in Software Engineering, p 48

Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for Java methods. In: ASE 2010, Ieee/acm International Conference on Automated Software Engineering, Antwerp, Belgium, September, pp 43–52

Sridhara G, Pollock L, Vijayshanker K (2011) Generating parameter comments and integrating with method summaries. In: IEEE International Conference on Program Comprehension, pp 71–80

Sun X, Liu X, Hu J, Zhu J (2014) Empirical studies on the NLP techniques for source code data preprocessing. In: EAST of ICSSP, pp 32–39

Winkler WE (1999) The state of record linkage and current research problems. Statistical Research Division Us Bureau of the Census 14

Zhong H, Zhang L, Xie T, Mei H (2009) Inferring resource specifications from natural language API documentation. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA, ASE '09, pp 307–318, DOI 10.1109/ASE.2009.94, URL http://dx.doi.org/10.1109/ASE.2009.94