An Investigation of Compound Variable Names Toward Automated Detection of Confusing Variable Pairs

Hirohisa Aman*, Sousuke Amasaki†, Tomoyuki Yokogawa†, Minoru Kawahara*
* Center for Information Technology, Ehime University, Matsuyama, 790–8577 Japan
Email: {aman, kawahara}@ehime-u.ac.jp
† Faculty of Computer Sc. & Systems Eng., Okayama Prefectural University, Soja, 719–1197 Japan
Email: {amasaki, t-yokoga}@cse.oka-pu.ac.jp

***Abstract*—A successful naming of variables is key to making the source code readable. Programmers may use a compound variable name by concatenating two or more words to make it easier to understand and more informative. While each compound variable name itself may be easy-to-understand, a collection of such variables sometimes makes a "confusing" variable pair if their names are highly similar, e.g., "shippingHeight," vs. "shippingWeight." A confusing variable pair would adversely affect the code readability because it may cause a misreading or a mix-up of variables. Toward automated support for enhancing the code readability, this paper conducts a large-scale investigation of compound variable names in Java programs to find quantitative criteria of the confusing variable pairs. The investigation collects 31,806,749 pairs of compound-named variables from 684 open-source Java projects and analyzes them from two different perspectives of name similarity: the string similarity and the semantic similarity.**

***Index Terms*—Confusing variable names, string similarity, semantic similarity.**

I. Introduction

Programmers usually declare and use variables in their programs. Variables are fundamental components of programs, and variables' names have significant impacts on code readability [1]–[3]. Well-chosen names would make the code readable and help developers understanding and reviewing the program smoothly. On the other hand, poorly-chosen names degrade the code readability: if we replace the variable names with unrelated (meaningless) strings, we can easily obfuscate our programs [4], [5].

To make a variable name easy to understand, many programmers use English dictionary words or well-known abbreviations for the name [6]. The usefulness of choosing fully spelled words or abbreviations for variable names has been empirically proved in the past [7], [8]. When the role of the variable becomes complex, it is helpful to use a *compound* name [9] that is organized by two or more words, like "positionOfPlayer." Because a compound name corresponds to a phrase describing the variable, many programmers and code reviewers can easily understand the variable's role. The positive effect provided by a compound name has also been proved in empirical studies [10].

However, there is also a negative effect of compound names on code readability. Tashima et al. [11] proposed the notion of *confusing* variable pairs, where two variables have highly similar names like "lineIndex" and "lineIndent." Due to their high similarity, they cause a risk of misreading or mixing up variables during the programming or code review. Thus, they may adversely affect the code readability even though each variable name is understandable. Aman et al. [12] reported an empirical study showing that confusing variable pairs are related to a deterioration of code quality. Hence, it is significant to detect confusing variable pairs for successful code readability management.

Although the previous studies [11], [12] measured the similarity between variable names using the Levenshtein distance [13], they missed the semantic perspective. For example, we may consider "stackNum" to be similar to "stackCount" because we can use the word "count" instead of "number" ("num") in many contexts. It is better to consider both the string and semantic similarity to examine the confusing variable pairs. This paper conducts a large-scale investigation of compound variable names to analyze the name similarity from both the string similarity and the semantic similarity perspectives as a preliminary work toward an automated detection of confusing variable pairs. The analysis result presents quantitative criteria to detect confusing variable pairs.

II. Compound Variable Name and Name Similarity

In this section, we describe the variable name of interest, the compound name. Then, we introduce the notion of confusing variable pairs and explain metrics for detecting those pairs.

A. Compound Variable Name

It is desirable that a variable's name clearly expresses its role in the program. To this end, programmers would choose a fully spelled word or its abbreviated form as a variable name. However, there are also cases that it is hard to express a variable's name by a single word because its role is complicated. For such variables, programmers tend to use compound names that consist of two or more terms (words or abbreviations). A compound variable name is produced by concatenating terms. The camelCase and the snake_case are the most popular naming styles [9]. In the camelCase, we join all terms and capitalize the initial character of the second or later terms to indicate the separation. In the snake_case, we indicate the separation of terms by the underscore ("_"). For example, if we make a name from "data file name," the

compound names in the camelCase and the snake_case can be "dataFileName" and "data_file_name," respectively.

Some programmers produce a compound variable name by concatenating two or more terms without any indication of separation, such as "filename"(file + name), because the element terms are simple and widely used. Although it is challenging to define the simple-concatenated compound name clearly, we regard a name as a compound variable name in this paper if we can split it into English dictionary words or well-known abbreviations[1].

B. Confusing Variable Pair

A compound variable name seems to be a phrase describing the role of the variable. Thus, it is understandable for many programmers and code reviewers without additional descriptions, such as the comments in the program. However, when we have two or more well-described compound names in our code fragment, they may cause a side effect on the code readability. If those compound variable names are similar to each other, they can be *confusing* variable pairs [11], [12]. Some programmers or code reviewers mix up those variables during the programming or reviewing activities, even though each compound variable name is understandable.

Figure 1 shows an example of the situation where a programmer encounters confusing variable pairs during their programming on Eclipse, an integrated development environment (IDE). In this example, the programmer is about to write "shippingWeight," and the IDE suggests candidates whose names are highly similar to each other. Because the first six candidates are the same-typed variables, the IDE would not warn even if the programmer selected a variable incorrectly. Moreover, suppose the programmer wrongly chose "shippmingMaxWeight" for it. In that case, the programmer and the code reviewers may not quickly find the mistake because "shippingWeight" and "shippmingMaxWeight" are highly similar to each other in terms of both the string similarity and the semantic similarity. Although the above case shown in Fig. 1 is just an example, it illustrates the risk of the variable mix-up caused by confusing variable pairs.

Here, we formally define the confusing variable pair below.

Definition 1 (confusing variable pair): We consider two variables v_1 and v_2 in a program P, and their names are *name*(v_1) and *name*(v_2). Let the line number intervals $S(v_1) = [b_1, e_1]$ and $S(v_2) = [b_2, e_2]$ be the scopes of v_1 and v_2, respectively; That is, v_1 is available between the b_1-th line and the e_1-th line in P, and v_2 is available between the b_2-th line and the e_2-th line. Let $type(v_1)$ and $type(v_2)$ be the data types of v_1 and v_2, respectively. We define the pair (v_1, v_2) to be confusing variable pairs if it satisfies all of the following conditions: (1) $S(v_1) \cap S(v_2) \neq \emptyset$, (2) $type(v_1) = type(v_2)$, and (3) *name*(v_1) and *name*(v_2) are similar to each other. □

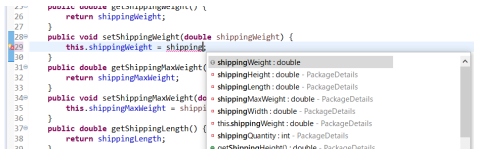


Fig. 1. An example of confusing variable names that we may encounter during our programming activity.

[1]We leverage GNU Aspell 0.60.6.1 to check words.

Condition (1) focuses on whether both variables are available simultaneously or not. If there is an overlap between their scopes, they can be candidates of a confusing variable pair. Condition (2) filters out different-typed variables because it is easy to find a mix-up case when the variable's type differs from the expected one; The IDE or the compiler would issue a warning. Although there are some combinations of different data types such that one type can be a substitute for another type, we adopt the above straightforward definition in this paper. Because the type compatibility checking requires a detailed and rich program analysis, it is costly in our large-scale investigation; it is our future work. Condition (3) remains to be clearly defined. To overcome it, we have to define the similarity between variables. There can be two different perspectives: the string similarity and the semantic similarity. We describe these similarities in Sections II-C and II-D.

C. String Similarity

When two variable names have a common part (substring), they might be similar; The larger the common part is, the more similar they look. For example, a variable name "shippingHeight" looks more similar to "shippingWeight" than "productHeight." We can quantify such a string similarity by focusing on how many characters we should edit to convert one name to another. A character edit is one of character addition, deletion, and substitution. Then, the least number of character edits to convert can be an index of string dissimilarity, referred to as the *Levenshtein distance* [13].

We can quantitatively compare the difference of the similarity in the above example as follows. For the sake of convenience, we denote the Levenshtein distance between $name_1$ and $name_2$ by $d_L(name_1, name_2)$. Then, we obtain d_L("shippingHeight", "shippingWeight") $= 1$ and d_L("shippingHeight", "productHeight") $= 8$, where, for the first pair, we can convert $name_1$ to $name_2$ by only substituting "H" with "W," i.e., the least number of required character edits is one; On the other hand, for the second pair, we need eight-character substitutions to convert $name_1$ to $name_2$.

Although d_L reasonably measures how two names look different, the length of the name may also affect the similarity evaluation. For example, while the following two pairs have the same Levenshtein distance ($d_L(\cdot, \cdot) = 1$), the similarity level does not look identical: ("fileA", "fileB") and ("distanceBetweenAandB", "distanceBetweenAandC").
To take account of such a difference as well, we consider the following normalized Levenshtein distance, nd_L:

$$nd_L(name_1, name_2) = \frac{d_L(name_1, name_2)}{\max\{\ \lambda(name_1), \lambda(name_2)\ \}}\ ,$$

where $\lambda(name_k)$ is the length of $name_k$ (for $k = 1, 2$). Then, we can successfully reevaluate the above example pairs: nd_L values of ("fileA", "fileB") and ("distanceBetweenAandB", "distanceBetweenAandC") are 0.2 and 0.05, respectively.

Because the distance is an inverse measure of similarity, we use the following Levenshtein similarity, sim_L, in this paper:

$$sim_L(name_1, name_2) = 1 - nd_L(name_1, name_2) \ .$$

The range of sim_L value is $[0, 1]$. The higher value the pair has, the more similar they are.

D. Semantic Similarity

There have been techniques for quantitatively representing the semantics of words in natural language processing studies: word embeddings. An embedding is a numerical vector representation of a word. Word2Vec [14] is one of the most popular models to produce word embeddings using the neural network. It learns the associations of words in the training data (texts), and Word2Vec uses the trained network to produce the corresponding word vectors. Because semantically similar words are likely to appear in a similar context, the corresponding vectors can also become close in the resulting vector space. That is, we can evaluate the semantic similarity between words using the closeness between the corresponding vectors. To produce the document (sequence of words) embedding rather than the word embedding, an extended version of Word2Vec has been developed: Doc2Vec [15]. We can also evaluate the semantic similarity between documents using Doc2Vec.

By splitting a compound variable name into its element words, we can regard the variable name as a document. Hence, we can also measure the semantic similarity between variable names using Doc2Vec through such a name splitting. We explain the steps to produce document vectors for compound variable names below.

(1) **Name Splitting**: We split a compound variable name by the camelCase or the snake_case[2]. For a simply-concatenated name like "filename," we consider all available patterns of splitting into two terms[3] and check if both terms are in a dictionary or not.

(2) **Preprocessing**: We decapitalize all words to avoid the difference in letter case affects the vectorization. After that, we perform the stemming of the word to uniform the word styles because a word can appear in different styles like "list," "lists," "listed," and "listing." Furthermore, we sometimes encounter a variable name using a number such as "inputBuffer2." Although the number is a part of the name, it would not be essential for considering the meaning of the variable name. To avoid any impact caused by such a number, we replace all numbers appearing in a compound variable name with the special token "<num>."

(3) **Vectorizing**: After the above processings of variable names, we train the Doc2Vec model and vectorize those names. By computing the similarity between vectors, we can obtain the degree to which two names are semantically similar. In this study, our metric of similarity between vectors is the cosine similarity. The range of similarity is $[-1, 1]$; The higher value the variable pair has, the more similar they are.

□

Through the above steps, for instance, we would be able to automatically judge that "stackNum" is similar to the following names: "numStack," "countStack," and "stackCount."

[2]We used the regular expressions "`[a-z][A-Z]`" and "_" as our simple delimiters.

[3]We split a name so that both the lengths of split names are longer than two characters to avoid many wrong extractions of "a" or "an."

III. Quantitative Investigation

We conducted a large-scale investigation of compound variable names. In this section, we report and discuss the results.

A. Aim

Toward an automated detection of confusing variable pairs, we need to develop reasonable criteria regarding the variable name similarity. By examining the name similarity trends (distributions) in the actual programming world, we can see a practical threshold level of similarity to judge whether a pair of compound variable names is confusing or not. Our investigation aims to find such criteria in terms of string similarity and semantic similarity.

B. Data Collection

We collected 1000 repositories of open-source Java projects from GitHub. We selected them by searching projects with the keyword "Java" in the descending order of the "stars" score. The number of collected projects (1000) is from the limitation of GitHub API. We developed a program analysis tool, JavaVariableScopeExtractor, that analyzes Java source files to extract variables with their data type and scope information. Because our tool supports only Java, we had to limit our investigation into Java projects.

We performed our data collection in the following procedure; Table I shows our environment.

(1) We obtained project information via GitHub API and made clones of Git repositories for each project.

(2) We analyzed all Java source files and extracted all variables using our tool. Variables include local variables, methods' formal parameters, and classes' fields. In our analysis, we excluded programs for testing, demo, or documentation because they are not the primary products of the development.

(3) We checked the names of all variables and picked up compound variable names from them. Then, we built the set of variable pairs where two variables are simultaneously available

TABLE I
Data Collection Environment and Hyperparameter

Item	Description
CPU	Intel Core i5-4570 3.20GHz
Memory	16GB
OS	Linux 3.10.0
Python	Python 3.6.8
stemmer	PorterStemmer on nltk 3.5
Doc2Vec	Doc2Vec on gensim 3.8.3 (vector_size=300, window=2, dm=1, min_count=1, epochs=100)

within a scope, and their data types are the same.
(4) For each variable pair, we measured the string similarity and the semantic similarity to examine if it can be a confusing variable pair or not.

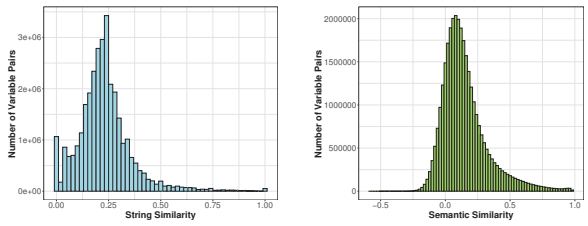
C. Results

As a result of our data collection, we successfully analyzed 60,908 Java source files from 684 projects[4]. The remaining 316 projects corresponded to any of the following ones: (a) they have no Java source files, (b) all source files are to be excluded ones such as test programs, or (c) our analysis tool failed to parse the source files because multi-byte characters appeared in the variable names.

From the above source files, we collected 1,377,219 variables, and 563,445 out of them (40%) were the variables with compound names. By checking their types and scopes, we found 31,806,749 pairs as candidates for confusing variable pairs in this study.

We measured the string similarity and the semantic similarity for each pair. Figure 2 and Table II show the distributions of similarity scores and the descriptive statistics. Because there are the variable pairs where two names are the same[5], the maximum string similarity is 1.0; Notice that Doc2Vec's vector production has an instability caused by random numbers, so the maximum semantic similarity is not equal to 1.0 even if the names are the same.

From the results of similarity measurement, we can see that most variable pairs have relatively low similarity in terms of both the string similarity and the semantic similarity: the medians of the string similarity and the semantic one are 0.222 and 0.115, respectively. In other words, there are not many variable pairs such that they look confusing (highly similar) in the actual Java programming world. We believe such a trend is natural and consistent with our programming practice, i.e., we are likely to avoid using the confusing variable pairs in our programs.



(a) string similarity (b) semantic similarity

Fig. 2. Histograms of similarity scores.

TABLE II
DESCRIPTIVE STATISTICS OF SIMILARITY SCORES

	Min	25%	50%	Mean	75%	Max
String	0.000	0.159	0.222	0.230	0.281	1.000
Semantic	–0.572	0.034	0.115	0.151	0.221	0.988

[4]Our dataset is available at https://bit.ly/3zH8HEd.

[5]For instance, one variable is a class's field, and another is a local variable in a method.

D. Discussion

To detect confusing variable pairs while considering the similarity distribution, we use the mean (μ) and the standard deviation (σ). When a value is greater than $\mu + 3\sigma$, we can regard it as an extremely high value. In the resulting dataset, μ and σ are $\mu_{str} = 0.230$, $\sigma_{str} = 0.130$, $\mu_{sem} = 0.151$, and $\sigma_{sem} = 0.179$, where the subscripts str and sem signify the string similarity and the semantic one, respectively. Then, we obtain thresholds of similarity as: $\mu_{str} + 3\sigma_{str} = 0.619$ and $\mu_{sem} + 3\sigma_{sem} = 0.686$. By using the above thresholds, we consider criteria of confusing variable pairs. To reasonably combine two measures, we need to pay attention to their correlation. We examined the correlation between the similarity scores using Spearman's ρ, and we confirmed no strong correlation between them ($\rho = 0.177$). Hence, we can integrate the above criteria in an orthogonal way, i.e., our criteria are:

- Is the string similarity is higher than 0.619?, and
- Is the semantic similarity is higher than 0.686?

As a result, 348,909 pairs and 383,593 pairs have the string similarities and the semantic similarities higher than the above thresholds, respectively (except for the same name pairs); 161,224 pairs (0.5%) satisfied both of two criteria. Table III shows samples from our data: No.1–4 are the pairs satisfying both of the above criteria, and No.5–8 are the ones satisfying only one of them. Pair No.1 looks like a similar pair, but it may not be hard to distinguish them; We may say its confusing level is relatively "Low" because both similarity scores are close to the thresholds. However, pair No.4 *must* be a confusing variable pair; We would say its confusing level is "High."

Pairs No.5–6 are samples of high string similarity, but low semantic similarity; Pairs No.7–8 are opposite cases, i.e., pairs with low string similarity but high semantic similarity. The former samples may look like confusing pairs as the previous studies [11], [12] because of their high string similarity scores. Furthermore, our proposal covers the latter samples as well. Although they may not look like similar names in terms of string similarity, there is a risk of mixing those variables while considering the variables' roles because of their semantically similar names.

TABLE III
EXAMPLES OF STUDIED VARIABLE PAIRS

No.	Variable Name Pair	Similarity String	Similarity Semantic
1	sentinelConnectionTimeout sentinelSoTimeout	0.640	0.758
2	lookupDefineClassMethod lookupDefineClass	0.739	0.889
3	singleFlatMapObservable singleFlatMapHideObservable	0.852	0.824
4	btGeneric6DofSpringConstraintDataName btGeneric6DofSpring2ConstraintDataName	0.974	0.903
5	getAlgorithmConstraints setAlgorithmConstraints	0.957	0.131
6	GradientColor_android_centerY GradientColor_android_endY	0.862	0.398
7	PUBLIC_KEY publicKey	0.100	0.921
8	edenUsed survivorUsed	0.333	0.897

From these results, our approach seems to detect confusing variable pairs successfully. Because the above thresholds of similarity is an example using the notion of "$\mu + 3\sigma$," we can leave them as user-adjustable parameters when implementing an automated detection tool. To decide more practical thresholds, we need to analyze the relationship of the above similarity scores with the manual evaluations. Toward a better threshold setting, we plan to conduct a questionnaire study regarding the variable names' confusing level in the future.

E. Threats to Validity

1) Internal Validity: The way of splitting variable names can be a threat to internal validity. Although we split compound names using simple regular expressions, they might cause inappropriately split names in our dataset. A richer splitting tool[6] might be a better choice in our investigation.

We measured the semantic similarity using the Doc2Vec model. Because its performance depends on the hyperparameters, their setting can be our threat to internal validity. To mitigate this threat, we preliminary experimented with various combinations of two major hyperparameters, vector_size and epochs. Because the Doc2Vec model steadily produced almost the same vectors for the same names when we set vector_size=300 and epochs=100, we used them in our study. However, there might be a better setting, and we might miss a more proper model for evaluating semantic similarity.

2) Construct Validity: We adopted the Levenshtein similarity as our measure of the string similarity. Moreover, we used the cosine similarity between the document vectors obtained by Doc2Vec, to evaluate the semantic similarity. Although these measures are widely used ones, there might be other better ways of measuring those similarities. To ensure that our measures properly quantify the similarity of interest, we need to prepare a dataset of variable pairs where the similarities are evaluated manually. Because it is costly and not easy work, we would discuss it continuously as a future issue.

3) External Validity: Although we conducted a large-scale investigation and analyzed many variable names in various programs, it is limited to open-source Java projects. Other projects using other languages or commercial projects may lead to different results. Because there are language-specific features or practices in naming variables, we plan to perform other investigations on other language projects in the future.

IV. Conclusion and Future Work

In this paper, we focused on compound variable names appearing in programs. Although compound names themselves are easy to understand, there is a risk that they can be confusing variable pairs when their names are highly similar. Because the confusing variable pairs may adversely affect the code readability, it is better to detect such pairs automatically. Toward the development of an automated detection tool, we conducted a large-scale investigation of compound names.

In our study, we collected 31,806,749 variable pairs from 684 open-source Java projects. Then, we measured the name similarity of them from two different perspectives: the string similarity and the semantic similarity. We used the Levenshtein distance and the Doc2Vec as our methods for measuring those similarities. As a result, we found that we may use 0.619 and 0.686 as the thresholds of the string similarity and the semantic similarity to detect confusing variable pairs, respectively. Note: Our dataset is available at https://bit.ly/3zH8HEd.

Our future work includes (1) developing an automated detection tool using our findings, (2) conducting a questionnaire study on practical thresholds of similarity scores, (3) further investigations on other software projects using languages other than Java, and (4) considering the ambiguity of variable names from the perspective of linguistic antipatterns [16].

[6] e.g., Spiral, https://github.com/casics/spiral

Acknowledgment

This work was supported by JSPS KAKENHI #20H04184, #21K11831, and #21K11833.

References

[1] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Softw. Q. J.*, vol. 14, no. 3, pp. 261–282, Sept. 2006.

[2] D. E. Knuth, *Selected Papers on Computer Languages*, CSLI Lecture Notes. Stanford, CA, 2003, no. 139.

[3] T. M. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. N.J.: Wiley Publishing, 1996.

[4] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques," *Empir. Softw. Eng.*, vol. 19, no. 4, pp. 1040–1074, Aug 2014.

[5] D. Low, "Protecting java code via code obfuscation," *Crossroads*, vol. 4, no. 3, pp. 21–23, Apr. 1998.

[6] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proc. 18th Annual Psychology of Programming Workshop*, Sept. 2006, pp. 53–67.

[7] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems & Softw. Eng.*, vol. 3, no. 4, pp. 303–318, Dec. 2007.

[8] G. Scanniello, M. Risi, P. Tramontana, and S. Romano, "Fixing faults in c and java source code: Abbreviated vs. full-word identifier names," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 2, pp. 6:1–6:43, Jul. 2017.

[9] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empir. Softw. Eng.*, vol. 18, no. 2, pp. 219–276, Apr. 2013.

[10] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proc. 26th Int. Conf. Program Comprehension*, May 2018, pp. 31–40.

[11] K. Tashima, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "Fault-prone java method analysis focusing on pair of local variables with confusing names," in *Proc. 44th Euromicro Conf. Softw. Eng. & Advanced App.*, Aug. 2018, pp. 154–158.

[12] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "Empirical study of fault introduction focusing on the similarity among local variable names," in *Proc. 7th Int. Workshop Quantitative Approaches to Softw. Quality*, Dec. 2019, pp. 3–11.

[13] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge: Cambridge University Press, 1997.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, Sept. 2013.

[15] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, May 2014.

[16] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: what they are and how developers perceive them," *Empir. Softw. Eng.*, vol. 21, no. 1, pp. 104–158, Feb. 2016.