Say my name! An empirical study on the pronounceability of identifier names

Remo Gresta
Federal University of São João del-Rei
São João del-Rei, Brazil
remoogg@aluno.ufsj.edu.br

Elder Cirilo
Federal University of São João del-Rei
São João del-Rei, Brazil
elder@ufsj.edu.br

ABSTRACT

Identifiers represent approximately 2/3 of the elements in source code, and their names directly impact code comprehension. Indeed, intention-revealing names make code easier to understand, especially in code review sessions, where developers examine each other's code for mistakes. However, we argue that names should be understandable and pronounceable to enable developers to review and discuss code effectively. Therefore, we carried out an empirical study based on 40 open-source projects to explore the naming practices of developers concerning word complexity and pronounceability. We applied the Word Complexity Measure (WCM) to discover complex names; and analyzed the phonetic similarity among names and hard-to-pronounce English words. As a result, we observed that most of the analyzed names are somewhat composed of hard-to-pronounce words. The overall word complexity score of the projects also tends to be significant. Finally, the results show that the code location impacts the word complexity: names in small scopes tend to be simpler than names declared in large scopes.

1 INTRODUCTION

The software maintenance activity can easily exceed the cost and time of other activities, such as system implementation. Therefore, it is well-recognized as the most expensive activity in software development [10]. As high-quality names might ease the developer's source code understanding, the software maintenance cost can be reduced by choosing intention-revealing names, as advocated in [16, 20]. Indeed, as Booch suggests [3], software should be described via natural language in the earlier stages. High-quality nouns should to be used as a starting point to design classes, and verbs used to describe operations. [3]. There are different means to achieve high-quality names; by using: intention-revealing words; searchable words; domain concepts, and pronounceable words [17]. Especially, pronounceable names can improve the discussion in code review sections (e.g., an experienced developer helping a newcomer). With easy-to-pronounce names, the discussion can happen with less tension and be more productive [17].

To investigate whether identifier names are in general easy to pronounce we carried out an empirical study in which we analyzed 1,421,607 names from 40 open-source projects. We performed repository mining to investigate three research questions: **RQ_1:** *How are names in Java projects compared with hard-to-pronounce words in English?*; **RQ_2:** *What is the complexity score of names in Java projects?*; **RQ_3:** *Is there a variation in name complexity regarding identifier location in the source code?.*

To answer the research questions, we use three phonetic algorithms: Soundex [19]; Metaphone [18], and NYSIIS [21]. They take as an input a word and return a code representing the respective phonetic encoding. These algorithms have been used in many applications, such as spell checkers (to find a word with similar phonetic encoding to an incorrectly-written one) and database search algorithms. To determine the pronunciation relationship among the analyzed names and hard-to-pronounce words, we calculated the phonetics distance among them using the Jaro Distance [4] and the Match Rate Index [13]. Finally, to quantify the overall word complexity in the analyzed projects, we used the Word Complexity Measure [22]. This measurement takes as input the phonetic encoding of a word and calculates the respective complexity based on word patterns, syllable structures, and sound classes.

The remainder of this paper is organized as follows. The Section 2 presents the background on naming practices. Section 3 details how we carried out our study. The Section 4 outlines the results of our empirical study and provides a general discussion. Section 5 describes the threats to the validity. Finally, Section 6 presents some concluding remarks.

2 BACKGROUND

In this section, we highlight the role identifier names play in code comprehension. We also present some conceptual aspects about phonemes and the phonetic algorithms used in this study.

2.1 Identifier Names

Identifier names play an important role in software development since they can be seen as the most fundamental source of information in the source code [7]. Therefore, they impact not only code comprehension but also the overall code quality [6]. First, names are essential to represent real-world concepts in the source code [20]. Concepts can be represented in the source code as an object, single variables or parameters. Names chosen to represent concepts in the source code should be concise and consistent [9]. Second, high quality names can also be helpful as source code documentation. Terse names indeed do not contribute to source code discussions neither provide information about their purpose. Therefore, many researchers and practitioners emphasize the importance of using descriptive and meaningful identifier names [2]. Interestingly, the length of identifier names can also impact the code quality: the prevalence of very short names has a strong effect on fault-proneness [14].

As mentioned, identifier names must also be pronounceable [17]. It is impractical to discuss source code snippets composed of words that programmers cannot pronounce in a code review session. Therefore, pronounceable names positively contribute to source code discussion and code review sessions. However, as stated by [15],

abbreviations (often hard-to-pronounce) are prevalent in proprietary and open-source code. This fact indicates that newcomer developers need to learn and comprehend abbreviations, increasing the training cost. Complementarily, single-letter names and abbreviations also impose an increase in defect discovery time [11], resulting in software that is harder to maintain and evolve.

2.2 Phonetic Encoding

A phonetic algorithm indexes words by their pronunciation, i.e., they provide an index that operates from the sound of words. There are several phonetic algorithms, most of which were developed for the English language. The Soundex is the most well-known algorithm for creating phonetic indices. It is available in many database management systems (e.g., Oracle, MySQL). The algorithm considers the letters in a word and associates them with numbers, transforming a word into a code with four elements: an initial letter; and three numbers [19]. The New York State Identification and Intelligence System (NYSIIS) algorithm, in contrast to Soundex, tries to capture differentiation in pronunciation by encoding similar sounds [21]. Finally, the Metaphone algorithm also encodes words by reducing them to consonant sounds (16 sounds) [18].

We can use some algorithms to measure the similarity between two encoded-words (string distance algorithm). For example, the Match Rating Approach (MRA) is a phonetic algorithm with separate encoding and comparison rounds. Since the MRA encoding may return inconsistent encoding for similar words, we can use a string distance algorithm to compare the MRA-encoded words in a second round. We can also employ the exact two-stage process to the encoded words returned by the previously mentioned phonetic algorithms. In this case, the Jaro Distance algorithm is an alternative to calculate the distance between the encoded words by computing the number of insertions, deletions, and substitutions to get from encoding to another (also called edit distances) [21]. This algorithm returns a number between 0 (exact match) and 1 (entirely dissimilar), representing the dissimilarity between strings.

2.3 Word Complexity Measure

The Word Complexity Measure [22] is an algorithm designed to give words a representative complexity score. The algorithm uses word patterns, syllable structures, and sound to calculate the complexity score of a word [22]. Therefore, a word that gains a high score contains more complex morphology structures that are harder to pronounce.

3 METHODOLOGY

This section presents the research questions, the study's design, and goals.

3.1 Research Questions and Study Goals

This empirical study investigates names beyond their intention-revealing quality (which concept they aim to represent); it analyzes the impact of words on the pronounceability of identifier names. As discussed in Robert C. Martin Book (Clean code) [17] by Tim Ottinger, pronounceable names can positively contribute to source code discussions and improve the overall understanding between developers. To study the pronounceability of identifier names, we convert them into their respective phonetic encoding and calculate their pronunciation similarity among words considered hard-to-pronounce. Therefore, we settled the evidence what make identifier names hard to pronounce. We summarize the goals of the study in the following research questions:

- **RQ_1: *How are names in Java projects compared with hard-to-pronounce words in English?*** We examine some words considered hard to pronounce, to analyze if names chosen to be identifier names are close or not to those words.
- **RQ_2: *What is the complexity score of names in Java projects?*** Using the Word Complexity Measure, we seek to find the usual complexity score within a Java project.
- **RQ_3: *Is there a variation in name complexity regarding identifier location in the source code?*** This question proposes establishing source code locations (e.g., ATTRIBUTE, PARAMETER, VARIABLE) in which names can be more challenging to be pronounced.

3.2 Sample Selection

To obtain a broad view of the pronunciation of names in practice, we chose 40 Java projects to be part of the study, representing different domains of application, such as Natural Language Processing, Image Manipulation, Software Testing, Computer Vision. First, we selected well-known projects with several contributors and commits, for example: *Jenkins*, *Junit4*, *Spring-boot*, and *Tomcat*. To select additional projects to be part of our study, we picked other smaller projects from a curated list of awesome projects [1]. All of the projects are open-source and were downloaded directly from their Github repository. Overall, the analyze Java projects comprise more than 7 million lines of code. Therefore, we consider that we have selected a representative set of open-source projects.

3.3 Extracting Identifier Names

We extracted 1,421,607 names (i.e., ATTRIBUTE, PARAMETER, and VARIABLE) from the 40 collected projects. After applying the words tokenization, we get a total of 17,886 unique words. We examined the words of each extracted identifier and their distance to 40 hard-to-pronounce word to answer the RQ_1. To answer the RQ_3, we also collected the location in the code in which the identifier names appear. Analysing the location in the code where a name is declared may corroborate in the understanding of how developers choose words in diverse scopes.

To convert words into their proper phonetic encoding, we start out by extracting all the identifier names of the analyzed projects. To achieve this goal, we develop a script based on a tool called SrcML [2], which converts source code from different languages into an XML (Extensible Markup Language) representation [8]. In this XML file, the original properties of the code are kept inside different tags. For instance, there is a tag representing a class (<class>), following it, a tag name (<name>) stores the actual class name. The SrcML representation enables performing Xpath queries. We went through the XML files searching for the names and the ones found in the queries were stored in a database.

[1] https://java-lang.github.io/awesome-java/
[2] srcml.org

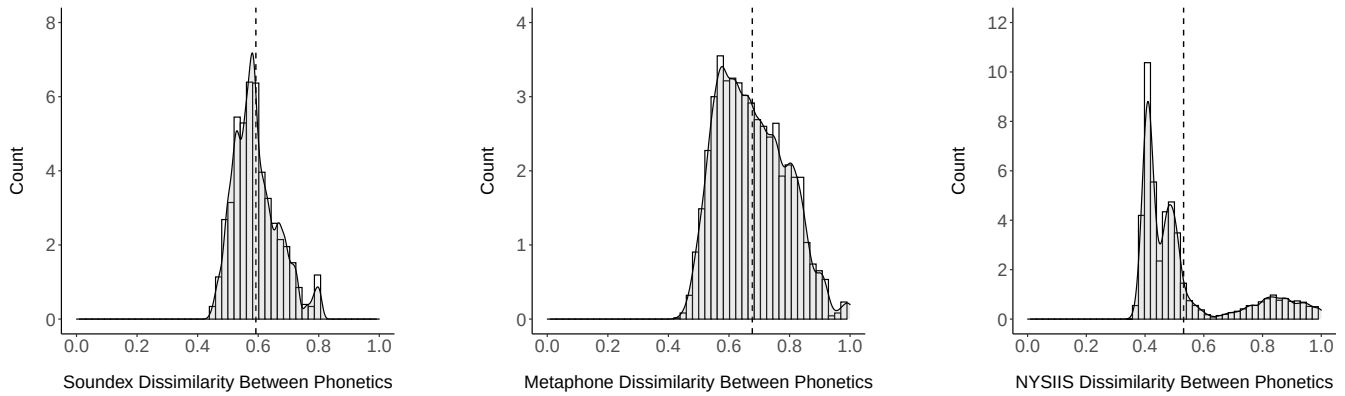


Figure 1: Dissimilarity Between Phonetics

3.4 Producing the Phonetic Encoding

The R library called Phonics [3], which implements phonetic algorithms (see Section 2.2), was employed to produce the phonetic encoding of the analyzed identifier names. The Soundex, Metaphone, and NYSIIS were the algorithms selected to produce the encodings. The words used to in the identifier names were used as input to the algorithms. Our motivation: during code review sessions, developers use the words to indicate a certain identifier; hence we chose to pass them as input to produce the phonetic codes.

We used the Jaro Distance algorithm (available in the R library `stringdist`) to compute the number of insertions, deletions, and substitutions required to transform one phonetic encoding into another. Therefore, in such a case, every word was compared with forty different hard-to-pronounce words in English [4]. Then, we calculated the mean of the dissimilarity between words. The exact process was replicated using the Match Rating Codex. In particular, this algorithm returns whether two words are similar or not regarding their pronunciation (a response of `True` or `False`). So, instead of computing the dissimilarity between words, we applied an OR operator in the returned Boolean values to resolve if an identifier has at least one challenging word to pronounce.

Finally, we calculated the WCM (complexity score) for all words in identifier names in all projects. Then, the mean of those scores was summarized for each project, representing the Project-specific WCM. We also derived the Location-specific WCM, representing the complexity score of identifiers declared in specific statements: ATTRIBUTE, PARAMETER, METHOD, FOR, WHILE, IF, and SWITCH. We used the Project-specific WCM to answer RQ_2 and the Location-specific WCM to answer the RQ_3.

4 RESULTS AND DISCUSSION

This section presents the results obtained in this study using the phonetic algorithms. We discuss the analysis and answer the research questions presented in Section 3.1.

[3] jameshoward.us/phonics-in-r
[4] www.thoughtco.com/hard-to-pronounce-words-4156950

4.1 RQ_1:How are names in Java projects compared with hard-to-pronounce words in English?

We employed the Jaro Distance algorithm to measure the dissimilarity between the phonetics produces using the Soundex, Metaphone and NYSISS. As mentioned in Section 3, we split the identifiers composed of more than one word and compared the word's phonetics with the phonetics of forty hard-to-pronounce words. The Figure 1 presents the resulting dissimilarities. As we can see, the dissimilarity measurement generally concentrates around the value 0.6, which means that the words part of the analyzed names are somewhat hard-to-pronounce (the Jaro Distance algorithm considers 0 as an exact match and 1 as completely dissimilar).

The Match Rate Approach and Jaro Distance measures the distance between word's phonetics. However, unlike Jaro, the MRA gives a `True` or `False` answer if two words have similar pronunciation or not. Whether the MAR algorithm considered at least one word as having a similar pronunciation, the respective identifier name received the value `True` (hard-to-pronounce) and `False` otherwise. We observed that the percentage of words that receive the value `True` is much higher than the `False` ones. The names having at least one hard-to-pronounce sub-word is 99% of the total. This result indicated a high number of words chosen to compose the analyzed names similar to hard-to-pronounce ones.

> Words chosen to serve as identifier names are somewhat hard-to-pronounce.

4.2 RQ_2: What is the complexity score of names in Java projects?

We used the Word Complexity Measure to answer the RQ_2. We calculated the WCM as the mean of the complexity of words in an identifier name. The overall score of a project was calculated by the mean of the scores of its identifier names. According to Table 1, the average word complexity varies from 2.04 (*Jtk* project) to 4.11 (*retrofit*). As observed in a study [22], the average complexity in a

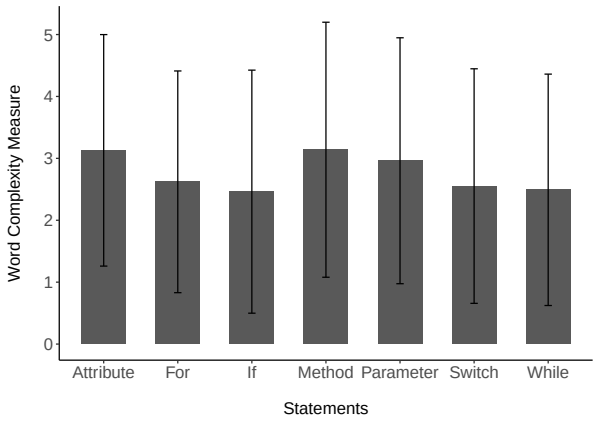
Table 1: Frequency of Special Characters

Project	Java LoC	Commiters	Commits	Complexity Score
aeron	108442	86	14409	3.62
androidutilcode	39030	32	1317	2.88
archunit	100276	49	1499	3.91
boofcv	650019	14	4520	2.61
butterknife	13279	97	1016	3.52
corenlp	581374	107	16280	2.95
dropwizard	74215	364	5789	3.91
dubbo	179477	386	4681	3.39
eventbus	8369	20	507	4.10
fastjson	179996	158	3863	2.70
glide	76418	129	2583	3.48
guice	72980	59	1931	3.40
hdiv	30631	11	1086	3.73
ical4j	24130	35	2303	3.07
j2objc	1810274	75	5284	2.74
jenkins	175150	654	31156	2.87
jtk	204105	9	1373	2.04
junit4	31242	151	2474	3.83
keywhiz	23337	32	1538	3.64
libgdx	272510	505	14661	2.76
litiengine	75877	20	3324	2.97
lottie-android	16258	102	1292	3.30
mockito	55751	220	5523	3.34
mpandroidchart	25232	69	2068	2.76
nutch	141710	43	3215	2.92
okhttp	48465	235	4848	3.97
orienteer	55681	12	2274	3.12
picasso	9136	97	1368	3.55
rest-assured	73511	105	2020	3.17
rest.li	523972	89	2617	3.48
retrofit	26513	152	1865	4.11
riptide	27072	18	2131	3.71
rxjava	468957	277	5877	2.77
spring-boot	343138	804	32096	3.97
tomcat	343703	61	23140	3.15
twelvemonkeys	99418	42	1334	2.64
unirest-java	15979	43	1603	2.83
webmagic	12926	40	1119	3.49
xchart	24406	50	1451	2.75
zxing	107064	109	3582	3.16
Total	7111470	5519	217869	3.26

set of 10 words spoken by adults is 2.8. Therefore, projects such as *retrofit* and *eventbus* possess a word complexity score superior to the average observed by [22]. This result might indicate the use of more complex words in some projects. On the other hand, the projects *jtk* showed the lowest complexity score among all the projects, and interestingly, this project also has the lowest number of committers.

The overall mean of scores is 3.26. This score is almost 1 point higher than the complexity of the words spoken by adults [22]. The majority of the analyzed projects received scores between 3-4. This result can indicate that developers tend to use words that are far complex than the average. However, as adults, developers seem to use a reasonable number of words that adults understand. Another observation is the low diversity in the number of words. We recognized only 17,886 unique words in the 1,421,607 extracted names. Therefore, this result might justify the high homogeneity in the WCM scores across projects (3 ± 1).

Figure 2: Complexity scores in different statements



> The word complexity measure in open-source projects, on average, is higher than in studies with adults.

4.3 RQ_3: Is there a variation in name complexity regarding identifier location in the source code?

Further, to answer the RQ_3, we investigated the Word Measurement Complexity over particular contexts (ATTRIBUTE, PARAMETER, METHOD, FOR, WHILE, IF, and SWITCH). The results are present in Figure 2.

The most complex identifier names appear as ATTRIBUTE, PARAMETER, and VARIABLE (complexity score of around 3). This result might indicate that in such contexts (ATTRIBUTE, PARAMETER, and VARIABLE), identifier names contain words that are more similar to hard-to-pronounce ones. In contrast, names present in relatively small scopes (e.g., FOR, WHILE, IF, and SWITCH) receive lower WCM scores. We suppose that, in such context, names are, in general, simple single-letter words. Indeed, some naming conventions [1] acknowledge the use of single-letter words to name a local temporary or looping variable. Therefore, identifiers declared in small scopes tend to have identifiers with less complexity than those declared in large scopes.

> The size of the scope influences the word complexity of the identifier names declared within them.

5 THREATS TO VALIDITY

Our study is subject to some threats to its validity. In this section, we present potential threats and how we tried to mitigate some of those issues.

External Validity. To mitigate threats concerning the generalization of our study, we selected a heterogeneous sample of 40 open-source Java projects. As our sample covers small code-bases (with less than 10K LoC) and large-scale ones (with over 100K LoC),

we consider the impact of this threat as minimal. However, given our sample size, we cannot control that our results do not reflect how name's pronounceability occurs in the wild. Moreover, to understand how hard identifier names are to be pronounced, we used a set of hard-to-pronounce words. Therefore, another potential threat is that these selected words are not representatives ones.

Internal Validity. A threat to the internal validity of our study comes from the number of names we analyzed in our study. To mitigate this threat, we collected 1,421,607. Additionally, another potential threat is how well the phonetic algorithm reflects extant the identifier names pronunciation. We tried to mitigate this threat by drawing from previous research, which has been extensively using such algorithms in practice.

6 CONCLUSION

According to Tim Ottinger (Clean code) [17], identifier names must persist pronounceable to empower developers in code review sessions. However, coming up with proper identifier names is challenging [5], and, as stated by [12], even though programmers have to name identifiers daily, it still entails a great deal of time and thought. This study investigated the pronounceability of identifier names in practice. We analyzed the identifier names using three phonetic algorithms (Soundex, Metaphone and NYSIIS) and also outlined the word complexity across projects and source code contexts (i.e., ATTRIBUTE, PARAMETER, METHOD, FOR, WHILE, IF, and SWITCH).

Our results based on 1,421,607 identifier names and 7,886 unique words from 40 open-source projects would seem to suggest the following:

- Words chosen to serve as identifier names are somewhat similar to other words considered as hard-to-pronounce.
- The word complexity measure in open-source projects is higher than in studies with adults.
- The programmer's naming practices are context-specific: more complex words seem to be more common in large scopes (ATTRIBUTE, PARAMETER, VARIABLE).
- We could benefit from including pronounceable naming practices in code reviews. The current practices do not address naming issues.

Our work highlights the need for further research on how naming practices are prevalent in source code and how better words can be chosen. Our long-term goal is to identify opportunities to rename identifiers and understand more about programmers naming practices. Finally, as future work, we plan to perform a study involving source code written in other programming languages (e.g., C and Python).

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *International Symposium on Foundations of Software Engineering*.

[2] Eran Avidan and Dror G Feitelson. 2017. Effects of variable names on comprehension: An empirical study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 55–65.

[3] Grady Booch, Douglas L Bryan, and Charles G Petersen. 1994. *Software engineering with Ada*. Vol. 30608. Addison-Wesley Professional.

[4] Leonid Boytsov. 2011. Indexing Methods for Approximate Dictionary Searching: Comparative Analysis. 16 (2011).

[5] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554.

[6] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 156–165.

[7] C Caprile and Paolo Tonella. 1999. Nomen est omen: Analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*. IEEE, 112–122.

[8] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 516–519.

[9] Florian Deissenboeck and Markus Pizka. 2006. Concise and consistent naming. *Software Quality Journal* 14, 3 (2006), 261–282.

[10] Nicolas Gold and Keith Bennett. 2004. Program comprehension for web services. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 151–160.

[11] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. 2017. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 217–227.

[12] Einar W Host and Bjarte M Ostvold. 2007. The programmer's lexicon, volume I: The verbs. In *International Working Conference on Source Code Analysis and Manipulation*.

[13] Deepjot Kaur and Navjot Kaur. 2013. A review: An efficient review of phonetics algorithms. *International Journal of Computer Science & EngineeringTechnology* 4, 5 (2013), 5068.

[14] Kimiaki Kawamoto and Osamu Mizuno. 2012. Predicting fault-prone modules using the length of identifiers. In *2012 Fourth International Workshop on Empirical Software Engineering in Practice*. IEEE, 30–34.

[15] Dawn Lawrie, Henry Feild, and David Binkley. 2007. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering* 12, 4 (2007), 359–388.

[16] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 3–12.

[17] Robert C Martin. 2008. Clean Code: A Handbook of Agile Software Craftsmanship.(2008). *Citado na* (2008), 19.

[18] Lawrence Philips. 2000. The Double Metaphone Search Algorithm. *C/C++ Users J.* 18, 6 (June 2000), 38–43.

[19] David Pinto, Darnes Vilariño, Yuridiana Alemán, Helena Gómez, Nahun Loya, and Héctor Jiménez-Salazar. 2012. The Soundex phonetic algorithm revisited for SMS text representation. In *International Conference on Text, Speech and Dialogue*. Springer, 47–55.

[20] Václav Rajlich and Norman Wilde. 2002. The role of concepts in program comprehension. In *Proceedings 10th International Workshop on Program Comprehension*. IEEE, 271–278.

[21] Chakkrit Snae. 2007. A comparison and analysis of name matching algorithms. *International Journal of Applied Science. Engineering and Technology* 4, 1 (2007), 252–257.

[22] Carol Stoel-Gammon. 2010. The Word Complexity Measure: Description and application to developmental phonology and disorders. *Clinical linguistics & phonetics* 24, 4-5 (2010), 271–282.