The Language of Programming: On the Vocabulary of Names

Nitsan Amit
Dept. Computer Science
The Hebrew University
Jerusalem, Israel

Dror G. Feitelson
Dept. Computer Science
The Hebrew University
Jerusalem, Israel

***Abstract*—Most of the text in a computer program is composed of the names of variables and functions. These names are selected by one developer, and need to be understood by others. This is similar to the role of words written in natural language. But there are several marked differences between the names in a program and the words in a book. First, names are frequently composed of multiple existing words, in an attempt to capture nuanced meanings and intents. Second, because of the use of multiple words, names can be rather long. Third, conventions may also allow names to be very short, and many single-letter names are used. But despite these differences, the general statistics of names are rather similar to the statistics of words. Like words, the distribution of names is close to a Zipf distribution. Also, popular names tend to be shorter than rarely used names. However, the underlying vocabulary if different. The composition of words leads to a more diverse vocabulary that can grow without bounds. But if we look at the individual words used in compound names, we find a rather limited vocabulary. These properties help explain the predictability of software, and how it can coincide with the large variability of names. It also suggests that it may be beneficial to model programs at the level of individual words rather than at the level of source code tokens.**

***Index Terms*—variable name, program lexicon, words distribution**

I. Introduction

When asked about the vocabulary of programming, most people think of programming language keywords: the basic types the language supports (`int`, `float`), the control structures (`if`, `while`), and so on. But the bulk of a program's text is not made of keywords but of names — mainly the names that developers give to variables, data structures, and functions [16]. It is the names which imbue programs with meaning. For example, consider the following pair of code snippets:

```
for x in c:
    if not r(x):
        x.s()
```

```
for student in class_members:
    if not ready_for_final_exam(student):
        student.study()
```

The first is meaningless in the sense that it could mean anything. The second can be read nearly as if it was English prose. The difference lies in the meaningful names given to variables and functions. They anchor the program in the domain, and identify the elements and operations which the program performs. We can therefore claim that it is the names which constitute the language of programming.

In studying the linguistic aspects of programming a natural place to begin is therefore the vocabulary of names. And given the emphasis in modern software development on self-explanatory readable code, an interesting initial question is whether the use of names in programs is statistically similar to the use of words in natural languages.

The study of word usage statistics is often associated with the name of George Kingsley Zipf. Following his investigation of word use frequencies in the 1930s, Zipf published two major findings[47]:

- The frequency of word usage is highly skewed. If the most frequently used word in a corpus is used k times, the second most frequently used word will appear about $k/2$ times, the third most frequent about $k/3$ times, and so forth. In general, the frequency of the r^{th} ranked word will be
$$f(r) = \frac{f(1)}{r}$$
This equation is now commonly known as "Zipf's law".
- A word's length is associated with its frequency ranking, with frequent words being shorter than infrequent words. Zipf called this "the principle of least effort" [48], based on the premise that uttering frequent words will require less effort if they are short.

We start our study by looking whether we can identify Zipf's law in programming language corpora. The data comes from popular active open-source Java projects hosted on GitHub. Extracting the names used in these projects and analyzing them reveals that the two elements of Zipf's findings indeed apply: the distribution is close to that given by Zipf's law, and more frequently used names tend to be shorter. In this sense program texts are similar to natural language texts.

However, we find that the distribution of name lengths in programs is different from the distribution of word lengths in natural language texts. For one thing, single-letter names are much more common than single-letter words. But more interestingly, names can also be much longer. The reason is that new names are created by concatenating multiple words together. While such compound words exist also in natural

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18).

languages, most notably in German, we find that they are much more prevalent and long in program code. Moreover, when the compound names are broken into their constituent words, we find that the vocabulary of these words is rather limited. So when looking at the structure of program names and natural language words we find that they are quite different.

Our contributions in this paper are

- To show that Zipf's law provides a reasonable description of the distribution of names, similar to words in natural language.
- To show that the distribution of name lengths in software is different from the distribution of word lengths in natural language corpora:
  - It has a mode at length 1 (single-letter names);
  - It has a heavier tail (non-negligible probability of very long names).
- To show an inverse correlation between a name's popularity and length. While a similar effect exists in words in natural language, the effect for names is stronger.
- To show that names tend to be composed of multiple words, much more so than the compound words that exist in natural languages.
- To show that the vocabulary of the atomic words used in compound names is rather limited.
- To identify mixed-style variable names, which combine camelCase style with snake_case style.

II. Background and Related work

There has been extensive research on Zipf's law for many years, attempting to provide reasons for this particular behavior of human language [40], [38]. A recurring finding is that the law as formulated by Zipf is only an approximation, and better statistical characterizations of the distributions of words in text can be found [7]. However, as a first approximation Zipf's law was also found to describe many phenomena in many fields [34], including, for example, the Internet, where it describes the distribution of popularity of web sites or the number of links to and from them among other things [1]. Another interesting result is that word lengths as described by Zipf are optimized for efficient communication [41].

Focusing on the linguistic perspective, one recurring criticism of Zipf's law is that the same relationship exists not only in human language but also in "monkey typing", that is the random typing of letters, one of which—the space—is used to delimit words [36], [33]. However, the nature of the words is completely different. Monkey typing creates a much larger vocabulary, with a geometric distribution of word lengths [21]. In real language words are but a small fraction of all possible letter combinations, and the distribution of word lengths is better described as log-normal.

Computer-related evidence of Zipf's law dates as far back as 1986. Researchers logged the aliases and user-defined commands used by students at the NASA Ames Research Center on a Unix command interpreter over a period of several months [18]. Collecting the results of all the usages and testing their rank-frequency ratio revealed that they indeed exhibited Zipf-like behavior. A more recent study by Dorin and Montenegro suggests applying Linguistic Economy to programming languages as a way to measure source code readability [17]. Linguistic Economy is the notion that words requiring less effort to speak are used more often than words requiring more effort. The study found that code files identified as unpleasant to review had more linguistically complicated identifiers in them than pleasant programs. Their identifiers' frequency/rank analysis of the source code produced results which support the data's compliance with Zipf's law. Halstead's Software Science is also based on the vocabulary of programs, with equations for size, complexity, and effort based on the numbers of operators and operands in the code [24]. But Halstead's formula relating vocabulary to code size does not hold for modern large-scale object-oriented software. Instead, a revised formula based on Zipf's law does [46].

Modern NLP methods go far beyond Zipf's law in creating and using statistical models for language processing [13]. At the same time, the notion that software shares statistical commonalities with natural language texts was suggested by Hindle et al. [27]. An important result was that most written code, just like most human utterances, is often repetitive and predictable. From this insight they went on to analyze the possibilities of modeling code with the same statistical language models used for natural language. This has led to a plethora of applications, most prominently source code generation [2]. Interestingly, recent work has also shown that code is actually *more* repetitive and predictable than natural language text [15]. This was attributed to the fact that writing code requires effort, like writing in a foreign language.

Our interest, in contradistinction, is less in the modeling of software and more in the linguistics of names as they relate to human developers. Most of the previous work in this area concerned the semantics of names, and its relation with the names' lengths. More than 30 years ago, Gorla et al. claimed that the effort to debug COBOL programs by students is minimized when name lengths are in the range of 10–16 characters [22]. In more recent work Holzmann showed that names are becoming longer with time [28]; on the other hand, Binkley et al. hypothesize that the limit on effective name lengths has already been reached, because longer names tax programmer memory [11]. Aman et al. show that length correlates with scope: in long functions (above 42 lines, which is the 90th percentile) single-letter names drop from 17% to 5%, and compound names grow from 34% to 51% [3].

Of special interest are compound names composed of multiple words. For example, Schankin et al. showed that such names contribute to comprehension [43]. Feitelson et al. presented a model of how concepts — and the words which represent them — are chosen when names are created [20]. Deissenboeck and Pizka suggest rules for the consistent composition of compound names, in the interest of improved naming [16]. They also link the consistency of name usage to program comprehension. Arnaoudova et al. extend this work to the use of individual terms that appear as components in compound names, and show that their indiscriminate use is correlated with fault proneness [5]. Interestingly, in other work Arnaoudova et al. show the fluidity of naming: specifically, the semantics of names may actually change when variables are renamed during refactoring, typically as a result of replacing a word in the name [4]. Conversely, Peruma et al. find that most renamings add terms in a way that narrows the semantics of the name [39]. However, this was for method and class names, not for variable names.

The natural language that is most famous for its use of compound words is German. Very much like in programming, German compounds are often created "on the fly". In contrast with other languages that have compounds, and especially those that are created ad-hoc, German compounds are not connected by hyphens but are simply melted together into a new word. Several studies have looked into the creation of natural languages compounds, and different outcomes of integrating them into everyday speech [26]. There has been ongoing research in an attempt to overcome issues with perception and translation of these by NLP and AI models [9], [42], since German compounds contain no clear borders between their constituent words and so it is difficult to split them apart for analysis and translation. Other studies have looked into how long or complex compounds might hinder comprehension by humans [30], and how the meanings of popular individual words in them might help [31]. Conclusions from such studies might be relevant for applying similar models to code readability.

III. Research questions

As noted above, programming language keywords are not really the building blocks of programs. The expressiveness of programs lies in the class, method, and variable names chosen by developers. So it is interesting to investigate the process by which such names are generated, and the degree to which it is similar to the creation of new words in natural languages. However, the available data — that is, the names that developers select — pertains to the outcome of the process and not to the generative process itself. What we can do is therefore to compare the vocabulary of names to that of natural languages. Based on this observation, our research question are

1) Is the distribution of names in programs similar to the distribution of words in natural texts? In other words, does Zipf's law apply?
2) Does software exhibit an inverse relation between name popularity and length, as seen in natural languages?
3) Are the statistics of compound names in programs similar to those of compound words in natural languages?
4) When compound names are divided into their constituent words, what is the distribution of the core vocabulary of these words?

IV. Methodology

Our approach to answering the above questions was to select several open-source code projects, and to extract and analyze the identifiers used in them.

TABLE I
GitHub projects used in the study.

		Names	
Project	*Owner*	*Total*	*Unique*
dbeaver	dbeaver	23,734	3,660
gson	google	15,246	1,194
Hystrix	Netflix	41,080	3,162
jenkins	jenkinsci	194,617	13,827
mockito	mockito	30,703	2,961
mybatis-3	mybatis	42,812	3,281
playwright-java	microsoft	25,912	1,646
rebound	facebookarchive	941	215
Signal-Server	erdinctaskin	45,454	3,698
xxl-job	xuxueli	16,088	1,696
All together		436,587	29,604*

*The total number of unique names is smaller than the sum of unique names in all the projects due to overlap in popular names.

A. Selection of Projects

A natural source for code to analyze is the GitHub repository. However, not all projects on GitHub represent solid software engineering work. We therefore initially tried to use the list of "engineered software projects" created by Munaiah et al. [37]. This list[1] contains software projects that leverage sound software engineering practices in several dimensions such as documentation, testing, and project management. The complete list contains over 1.85 million projects, so we focused on projects which were classified as "engineered" by both the Random Forest and the score-based models, were larger than the average repository size, and were written in Java. Our decision to focus on large repositories stemmed from our research questions: We wanted to test the code corpora for similarity with natural language phenomena, which only reveal in their fullest extent when the data is large enough. Java projects were selected for the language's popularity, community, and for consistency reasons.

This attempt turned out to be unsuccessful for several reasons:

- The projects list contained many duplicates.
- Many of the projects on the list had been deleted or archived since the list was created.
- Many of the projects on the list were listed as Java projects, but in fact contained less than 70% Java files.

We therefore reverted to using existing filtering tools available on GitHub to manually select the projects. Filtering for the most popular and active (trending) repositories written in Java, we came up with the projects listed in Table I.

All the analyses described below were applied to these ten projects. We note, however, that initially we had two more projects: `Runelite` and `RxJava`. We decided to remove them from consideration because they turned out to be significantly larger than the others, and so they skewed the results. Also, the `Runelite` project had a strange distribution dominated by 10-letter identifiers, possibly originating in some resource files

[1]Available at https://reporeapers.github.io.

or constants. The implications are discussed in the threats to validity (Section VI).

B. Extracting the Identifiers

Once the projects were selected, we moved to extracting the identifiers and names from the code files. Only `.java` files were used for the research. Test files were omitted, as they contain a lot of boilerplate code and repetitive patterns which do not reflect naturally written code. We used the `JavaParser`[2] Java library to generate abstract syntax trees and extract identifiers from Java source files. This included all the "Simple Names", defined as names which do not contain namespace dots. Names originating from a dots-separated namespace were divided into their components before being counted (so name.space.dots would be counted as: name=1, space=1, dots=1). Names were tagged with their role in the program (e.g., `VariableDeclarator`, `MethodDeclaration`, `MethodCallExpr` and so on). Naturally, this process did not take into account any comments, keywords, and import statements. Related and similar words, e.g. `count` and `counter`, were left separate.

We debated whether we should include type declarations in the data collected. In analogy to counting words in natural language texts, we approached this by trying to define the grammatical role of types in code. We concluded that one might think of type specifiers as adjectives in natural language, describing the nature of the following noun. For instance, in the following code snippet[3] the type `JsonWriter` describes the following new variable `jsonWriter`:

```
JsonWriter jsonWriter =
    new JsonWriter(stringWriter);
```

At first, we believed the types should be omitted to avoid skewing the result with names that are external to the project. Variable and method names are by definition always originally selected by the code developer, but types are often imported from external libraries or are predefined by the language itself. For instance, `String`, the most common word we found in many of the selected projects, is in fact a class defined in Java's basic library, and is not defined originally in any of the projects themselves. Nevertheless, omitting the types meant also ignoring classes such as `JsonWriter` above, which is an organic part of the `Gson` project files. With these arguments in mind, we decided to retain the "Type as Adjective" analogy and keep all type declarations and usages in the data.

C. Names' Analysis

For each row in the extracted names data file, we added two additional columns: length and wordCount. The length column was simply the number of characters in the name. The wordCount column was generated using a function which recognized the identifier's style (camelCase and snake_case), and then split the identifier to its constituent words. For instance, the method identifier `getResultNumber` would have a length value of 15, and wordCount value of 3. The splitting function also handled uppercase abbreviations and mixed cases, such as `randomUUID` (2 words), or `sizeOf_kinfo_proc_32` (5 words). Underscores at the beginning or end of an identifier were omitted, so `_my_name_` was considered a 2-words identifier, consisting of `my` and `name`.

Lastly, the data rows were aggregated by different columns depending on the question we were trying to address, grouping by name, length, or project. In most analyses, the data was sorted by frequency in a descending order, and a rank column was added to the aggregated data.

D. Comparison to Natural Languages

In addition to the programming language corpora, we used both an English and a German corpus for comparison. The English language corpus we selected is COCA, the Corpus of Contemporary American English. This corpus was selected for its size, popularity and availability. It also proved to be a good match for our research since it offers a frequency list for the top 60,000 most frequent words in the corpus.

The German corpus was selected for comparison with compound words. Compounds are words which consist of more than one stem, and are the result of chaining together several other words to form a new one. Although compounds exist in English (afternoon = after + noon, blueprint = blue + print), it is rare to find compounds with more than two stem words. In German, on the other hand, compounds consisting of three to four stem words are not at all uncommon (Büroarbeitsplatz = Büro + arbeit + platz, kreisvolkshochschule = kreis + volks + hoch + schule).

In our research, we wanted to compare the splitting of identifiers to their stem words with splitting of German compounds to their stem words. As a basic data source we used The German Reference Corpus (Das Deutsche Referenzkorpus — DeReKo[4]). Their word frequency corpus contains the 100,000 most common words in their 50-billion-words corpus, along with their frequency.

Identifying compound words in German is hard because there are no syntactic indications: the constituent words are just concatenated together. Rather than trying to identify them ourselves we used GermaNet's dataset of German compounds [25]. This dataset contains 106,780 different compounds, each with a breakdown of "head" and "remainder". We created a recursive function which peeled off each head until it split every compound to its individual components, and then counted their total. For instance, if the word was Büroarbeitplatz, the first split was head = Büro and remainder = arbeitsplatz, and the second head = arbeit and remainder = platz, leading to a total of 3 words. Running the above function on the DeReKo corpus, we were able to achieve a similar wordCount column for a natural corpus, as we did for the programming language corpus.

[2]https://javaparser.org

[3]Example from Google's Gson project. Using types in names like this is pretty common [23].

[4]https://www.ids-mannheim.de/en/digspra/corpus-linguistics/projects/corpus-development

TABLE II
The 40 most frequent names in all the projects combined.

1 String	11 Class	21 Jenkins	31 size
2 e	12 add	22 p	32 protobuf
3 Object	13 type	23 File	33 item
4 T	14 getName	24 equals	34 context
5 IOexception	15 LOGGER	25 req	35 json
6 get	16 log	26 key	36 Level
7 name	17 length	27 append	37 toString
8 List	18 r	28 c	38 put
9 value	19 result	29 Integer	39 s
10 i	20 options	30 Map	40 Exception

V. Results

A. Name / Word Frequency

Observing the most frequent names in all the projects combined (see Table II), we see that native Java classes (`String`, `Object`, `IOException`, `List`, `Class`, `Integer`, and so on) dominate the top of the frequency list. This finding is not surprising, as these are the most basic classes in Java, making up most of the declared variables in any project. It is similar to the dominance of articles and prepositions in English (the, to, of, in, it, etc.). To check whether the inclusion of such type names affects our results we ran all the analyses again after excluding them; the results were largely the same. All the following results therefore include all names, including types.

A more interesting result is the significant representation of single-letter variables among the top names. These include not only the ubiquitous loop index `i`, but also `e`, `T`, `c`, `s`, `f`, `t`, and more. These seem to serve as a “programmers’ slang”, i.e. agreed upon practices when naming certain variables [8], [23]. For instance, exception variables are almost always called `e` and generic types are called `T`.

Some function names also appear among the most common names. These can be divided into two classes. The first is commonly used generic methods, such as `length`, `equals`, and `toString`. The second is getters and setters, and especially `getName`.

To compare the distributions of names and words we need to consider their frequency. Figure 1 shows Zipf plots, namely the relation between the words’ frequency and rank on double logarithmic axes, for all the names in all the projects we’ve analyzed, and for COCA. These are reasonably straight lines, indicating a power law as anticipated by Zipf. The dent at the top ranks of names is due to the extremely high frequency of the first word, `String`, and the more equal frequencies of the following few names. The second graph shows the 60,000 most frequent words in COCA. The curving at the end of the plot is typical when observing large natural language corpora from various speech and text origins. The vocabulary is ultimately limited, and so the frequency of words keeps rising when more sources are appended, but new ranks (that is, new unique words) are added less frequently [45]. In programming languages on the other hand, we speculate that this phenomenon does not exist because of the generative nature of identifiers, which places no limit on the corpus vocabulary. Similar results and explanations were given by Casalnuovo et al. on larger datasets; however, they looked at all tokens and not just names [15].

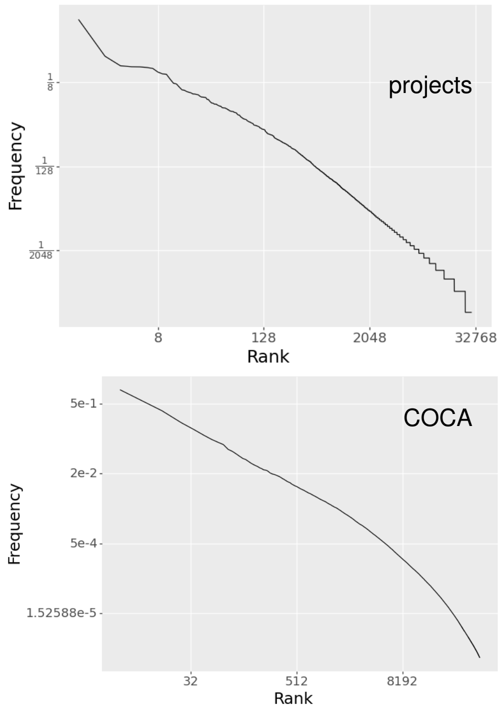

Fig. 1. *Zipf rank-frequency plots for all the names in all the projects and for English words in COCA.*

B. Name / Word Length

When analyzing the distribution of name or word lengths, one must distinguish between the distribution of unique names and words and the distribution of name and word occurrences. The distributions of unique names and lengths are similar in that they have a general bell-shape with a certain right tail. But their spread is different. For English (COCA) the most common word length is 7 letters long, and the longest observed is 21. In the German DeReKo corpus the most common length is 9, and the longest word is 27 letters long. And in the Java projects the most common name length was 12, and the longest observed was 75 characters long! (it was `internal_static_textsecure_ServerCertificate_Certificate_fieldAccessorTable`).

The distributions of name and word occurrences accentuate the differences in the tails (Figure 2). But they also exhibit marked differences for short lengths. In code, single-letter identifiers are widespread: single-letter names are the 7th most frequent name length in our projects, and in particular they are 3 times more common in comparison to two-letter names. Meaning, there is a special preference for single-letter names in programming, which is not caused by a general fondness of shorter names. In English, single letter words (a and I) are not uncommon, but less so than 2-letter words. In German

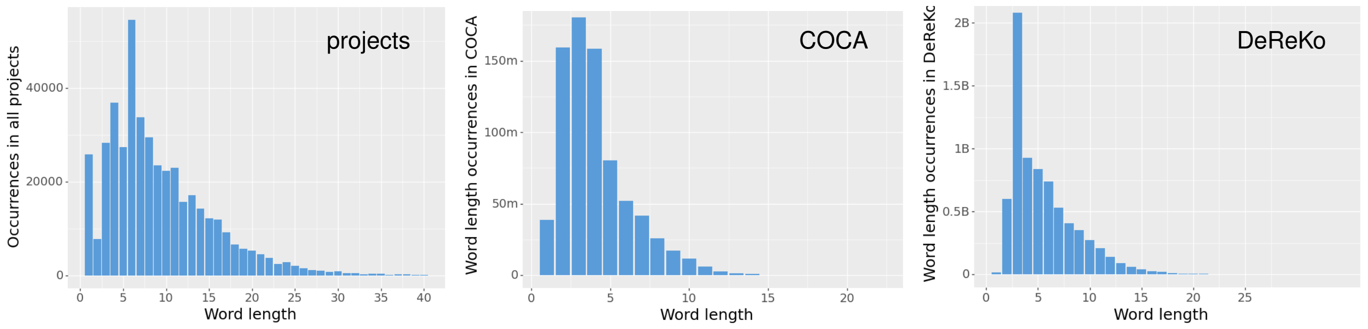

Fig. 2. *Lengths histograms for occurrences of names in all the projects (left), COCA English words (middle), and DeReKo German words (right).*

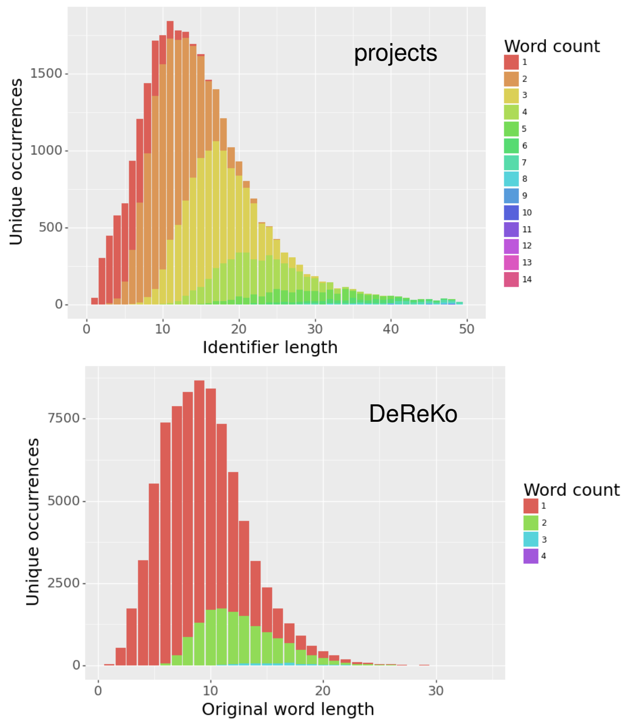

Fig. 3. *Top: name lengths histogram with word count breakdown for all projects. Bottom: German word lengths histogram with compound words breakdown, DeReKo.*

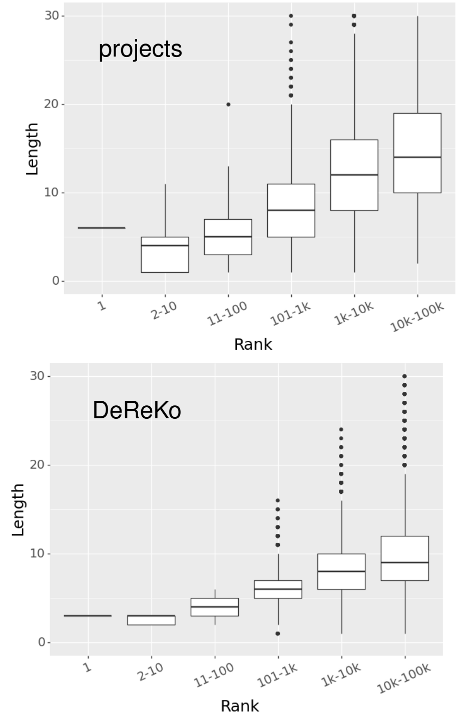

Fig. 4. *Identifier and word lengths by rank, Java projects and DeReKo.*

3-letter words predominate (e.g. der, die, und, von, mit, etc.), and there are no real single-letter words.

As noted above, software includes names that are much longer than any words in natural languages. This is due to the use of compounds composed of multiple words. Figure 3 shows histograms of names and words lengths, partitioned by the number of constituent words each one includes. As one might expect, as identifiers get longer, they tend to consist of more words. Testing the same breakdown on the German DeReKo corpus revealed a similar pattern, though not nearly as strong. While 3-words identifiers are quite frequent at lengths 10 to 25, and 4- and 5-word identifiers are also readily observed, German words consisting of 3 or more words are almost non-existent in the corpus.

C. Name / Word Length and Rank

An interesting observation made by Zipf was that word length is inversely correlated with frequency: the most frequent words tend to be short, and the infrequent ones long. We observe the same effect also in software entity names. To visualize this we partitioned the data into logarithmic bins by their frequency rank, and then created a boxplot representing the distribution of the words' lengths in each bin (Figure 4). There's quite some variation within each distribution, but looking at the percentiles the trend is clear: as names become less frequent in the project, the distribution of their lengths consistently shifts toward longer names. The only exception to this is the most frequent word in all the projects, `String`, whose popularity despite its non-minimal length we have already discussed.

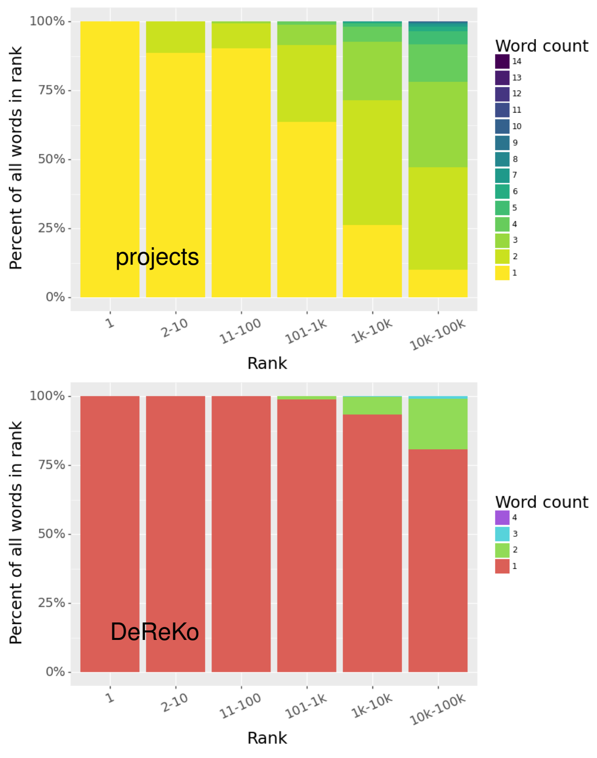

Fig. 5. *Word count by rank in identifiers from all projects, and in German compound words from DeReKo.*

As expected, this behavior was similarly observed in the German corpus, with the most common word being the singular male article “der”. One main difference between the plots is the variation in words lengths, which is much greater in the coding projects than the natural language corpus as mentioned above.

The inverse correlation of frequency and length matches the linguistics studies mentioned in the background and related work section, indicating that word lengths are optimized for efficient communication. This is similar to the efficient encoding of information, as in Huffman coding [29]: Languages too can be made more efficient by giving the most frequently used meanings the most concise symbols. Evolution through different organic social processes results in this effect in most spoken natural languages. It is therefore interesting to see that the same effect also exists in code, which is not subject to the same social processes (or, at least, to a much lesser degree). Using shorter identifiers can also be an attempt to improve program readability, by having shorter and more concise lines of code. In addition, shorter names exert less pressure on short-term memory [6], [11], [19], again indicating that there may be a benefit in using shorter names for more popular objects.

As seen in Figure 5, the greater length of low-frequency names is associated with being composed of more constituent words. This motivates an investigation of the core vocabulary of these constituent words next. In German words, on the other hand, it appears that many low-frequency words are longer despite not being compounds.

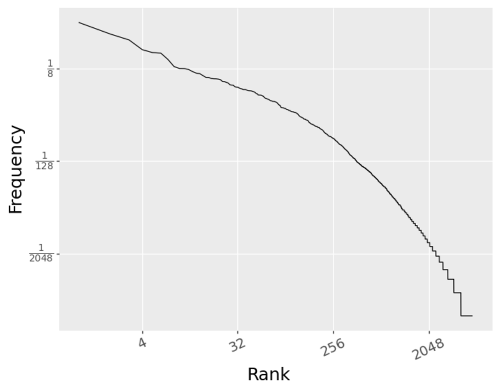

Fig. 6. *Zipf plot of the core-vocabulary of all names from all projects.*

D. *The Core Vocabulary of Names*

Using the name-splitting algorithm for finding the number of individual words which constitute a compound name, we can also extract these individual words and derive the “core-vocabulary” of the projects. This can then be fed into all the analyses performed above.

The core vocabulary is defined to be all the *individual words* used in names. Note that these words come from two sources:

- Non-compound names that are a single word to begin with;
- Words extracted from compound names by the splitting algorithm.

For example, if there were 10 usages of the identifier `getDisplayName` in the corpus, in the new data there were 10 more usages of `get`, of `display`, and of `name`, and no occurrences of `getDisplayName`. If there were 2 usages of `Hello` and 3 usages of `helloWorld`, in the new dataset there were 5 usages of `hello`. Note too that all words were lowercased to identify the different styles.

Looking at the new data and graphs generated with the new core-vocabulary presents a curious picture. First, in the initial data there were 29,604 distinct identifiers. In this case differently cased identifiers were considered to be different words, since case plays an important role in Java’s naming conventions, as they separate types and classes from variables and instances. After applying the algorithm mentioned above, there were only 5,329 distinct names left. The most common word in the corpus was still `string`, and words like `file`, `get`, and `key` climbed to much higher positions in the frequency chart.

This finding can have significant impact on language models for code. The use of long compound names causes severe problems for such language models, because of the large vocabulary of names that needs to be learned [32]. But if we break compound names into their constituent words the effect is reversed, and we have only a relatively small vocabulary to deal with. So learning a model of how compound names are created can replace the need to learn large corpora of code.

Returning to our study of the core vocabulary, the new generated Zipf’s plot now seemed to be much rounder, and

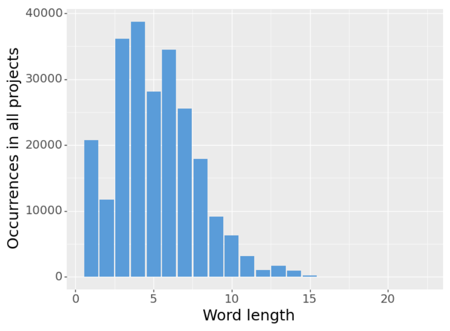

Fig. 7. *Core-vocabulary word lengths histogram, all projects.*

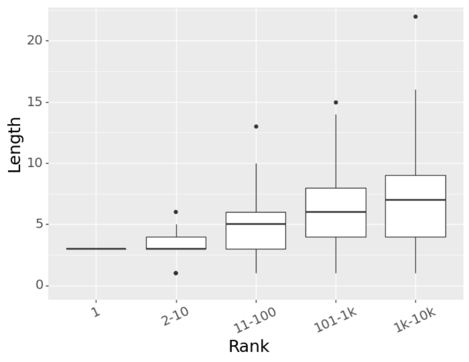

Fig. 8. *Core-vocabulary word lengths by rank, all projects.*

does no longer have a perfect straight line segment as before (Figure 6). There is also no significant curve at the top left corner of the graph as before. This is probably an indication that many less frequent words contain frequent words as components. As we've seen in Table II, only 4 of the 40 most popular names were compound names.

The words lengths histogram is now much closer to the histograms of natural languages we saw above (Figure 7 compared to Figure 2). Unlike the original names histograms, here we have a lot less variation in length, and most words are between 1-10 characters. The breakdown of the compound words brought the mean length of the identifiers much closer to the average length seen in the natural language corpora, with the exception of still seeing many single-letter words. Comparing with the distributions of word length occurrences from Figure 2, we can see somewhat more intermediate length words, of lengths 5–8 letters. This may reflect a need to represent various non-trivial concepts that are not so commonly encountered in natural language.

The change we see in words lengths is also reflected in the length by ranks boxplots (Figure 8). Breaking the data to its core vocabulary caused the effect seen in Figure 4 to be much reduced. It is still slightly noticeable that as words become less frequent their length tends to increase, but since the variation in length was very slight to begin with, the upward curve is now much shallower.

E. Mixed-Style Names

One of the religious wars of programming concerns the preferred naming style, especially whether to use camelCase or snake_case to join words when forming compound names. Several studies have been devoted to the possible effect of naming style on programmer performance [10], [44]. All these studies consider this as a dichotomy: it is assumed that programmers use either one or the other.

However, when looking at the names we collected in this study, we found that a mixed style is also used. This seems to be done to provide context to names. Thus levels of context are separated by underscores, and the name of each level may use camelCase if more than one word is needed. For example, the name `actions_menu_edit_ContentFormat` indicates that `ContentFormat` is one of the options in the `edit` option of a menu that is one of the possible actions. Additional examples from the same project context include `action_menu_transactionMonitor_notConnected` and `confirm_disconnect_txn_toggleMessage`. Additional examples from another project are `CreateViewCommand_ShortDescription`, `RunParameterDefinition_DisplayName`, and `PluginManager_newerVersionExists`.

The importance of this practice is that it provides another explanation for the construction of long names. Long names are not just a result of needing many words to describe a concept. They can be the result of building context, similar to the use of dot-separated namespaces. The availability of both the camelCase and snake_case styles enables developers to compose such names in a way that clarifies their structure. We note that this phenomenon was observed already by Butler et al. [12]. However, they were mostly concerned with the difficulties such names cause when trying to interpret names as phrases. Our view is that their use as a composite context indeed excludes a direct interpretation as a phrase. When a name is composed of multiple parts, each part should be interpreted as an independent phrase.

VI. Threats to Validity

A study of vocabulary depends on the words that are identified and used as data. Several decision we made can be debated, and alternative decisions may modify the results. However, to the degree that we checked such changes are expected to be minor.

We used a Java parser to extract names from the source code. An important decision is then exactly what classes of elements to include under the heading of "names". As discussed in Section IV-B we decided to include not only variable and function names but also type names. We also decided to include the namespace components of dot-separated names. We feel that these decisions correctly reflect the meaning of names in software, but other decisions are possible.

Another decision concerns names including numbers. Some of them should obviously be considered as part of a word. For example, Log4j in `Log4jImpl` is the name of a logging library, and UTF8 in `isValidUtf8` is an encoding. In others the number may be considered as a separate adjective, such as in `sha256`. There were also many names where it was not clear how to think of the number, for example `rememberMe2` or `with0`. Finally, in some cases the numbers seems to introduce spurious separations, for example in `p99_99` (percentile 99.99?). Given all these options, we elected not to try to untangle them, and to append numbers to their predeceasing word (if one exists) and count them together as one word. We note that in total there were 1166 names with numbers, which is just less than 4% of the unique names.

Another issue is the separation of compound names into their constituent words. Obviously due to their large number this cannot be done manually. We therefore rely on the fact that developers usually use case changes or underscores to signal word boundaries (camelCase/snake_case). However, we have also observed several identifiers with names like `INVOKESPECIAL` or `Shellinterpreter`, which do not include such signals. Hence, these were considered to consist of only one word according to our splitting algorithm. Using more sophisticated splitters would lead to more precise results [14].

The decision to collect data from only 10 projects, all in a single language, causes an obvious external validity issue. This is exacerbated by the fact that two projects which were initially selected to be included in the research were later omitted. The `RxJava` project turned out to be significantly larger than the other projects, and so its vocabulary and frequencies data overshadowed the data from the other projects combined. Using it would therefore lead to an even worse external validity issue, where the results actually reflect only one project. The `Runelite` project had an abnormal distribution where a big portion of its vocabulary was made up of 10-letter words, unlike any other project we've seen. For this initial research we chose to avoid such anomalies, but we must acknowledge that our results are definitely not universal.

Finally, it is worth mentioning some possible matters with the COCA dataset, which might have altered the results. The frequencies data provided by the corpus creators contains all the frequencies of the 5,000 most frequent words, and then data of only every 5th word. This doesn't change the Zipf's plot dramatically, but it might have an effect on the lengths histogram and length/rank boxplot. Another issue is that the dataset doesn't contain words that appear less than 20 times in the corpus, which might definitely influence all our results, as our whole assumption is that less frequent words tend to be longer.

VII. Conclusion

Phil Karlton is credited with the saying that "There are only two hard things in Computer Science: cache invalidation and naming things". And many agree that naming is indeed hard. It is also ubiquitous: developers constantly need to come up with new names for classes, methods, data structures, and variables. And these names need to be meaningful, that is, to convey the intended purpose to whoever will read the code later.

The creation of new words in natural languages is a continuous social process. People invent words and use them, perhaps with an explanation, and if they indeed fill a need and do so well, others will pick them up, and they eventually become part of the language. Developers don't have the benefit of such a process. When they invent a name, this is a final decision on how a variable or other structure will be referred to. And names that require an explanation (that is, a comment explaining what they mean) are not considered good names [35]. Good names should be immediately understood; they should explain themselves.

As a result developers cannot create completely novel names. Their names must use existing words, or at least be closely related to existing words. And when a name needs to convey specific details, the common approach to achieve this is to use several words in tandem. This leads to the creation of long compound names. At the same time, a programmer slang has emerged favoring short and even single-letter names for certain common uses.

Because of these distinct processes, the vocabulary of programming ends up being quite different from the vocabulary of natural languages. On one hand it allows the construction of infinitely varied new names, using long concatenations of existing words. On the other hand it uses a rather limited basic vocabulary, and some very short names are also very common. In these senses the language of programming differs from natural languages. However, these differences make names easier to understand and potentially more predictable, despite their length.

We note too that programming is similar to technical writing and not to prose. Programmers explicitly endeavor to write clear and readable code, and they are measured, inter alia, by their code's readability and modularity. They have no aspiration for literary sophistication. This preference becomes very clear when looking at programming projects' core-vocabulary, which is significantly smaller than that of natural languages corpora.

Our work suggests a number of directions for future research. One is to determine exactly how programmers come up with variable names and identifiers. What are their underlying motives and their priorities? Is it readability? Is it efficiency (shorter symbols)? Or perhaps they just conform to naming conventions (the "programmers' slang")?

Another appealing study topic could be to examine how does a program's vocabulary grow as the size of the project grows. Zipf's law implies that vocabulary growth is boundless, and indeed compounding words together facilitates the infinite creation of new names. But what are the dynamics of this process? We have only looked at static snapshots of projects, without investigating their progression through the version control system. It could be interesting to see if a project's vocabulary's growth rate resembles the rate observed in natural language corpora, when new sources or additional text from the same source are added.

Experimental Materials

The data and scripts we used are available on GitHub: `github.com/NitsanAmit/LanguageOfProgramming`

References

[1] L. A. Adamic and B. A. Huberman, "*Zipf's law and the Internet*". *Glottometrics* **3**, pp. 143–150, 2002.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "*A survey of machine learning for big code and naturalness*". *ACM Comput. Surv.* **51(4)**, art. 81, Jul 2019, DOI: 10.1145/3212695.

[3] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "*A large-scale investigation of local variable names in Java programs: Is longer name better for broader scope variable?*" In 14th *Quality of Inf. & Commun. Tech.*, pp. 489–500, Sep 2021, DOI: 10.1007/978-3-030-85347-1_35.

[4] V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Guéhéneuc, "*REPENT: Analyzing the nature of identifier renamings*". *IEEE Trans. Softw. Eng.* **40(5)**, pp. 502–532, May 2014, DOI: 10.1109/TSE.2014.2312942.

[5] V. Arnaoudova, L. Eshkevarti, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol, "*Physical and conceptual identifier dispersion: Measures and relation to fault proneness*". In 26th *Intl. Conf. Softw. Maintenance*, Sep 2010, DOI: 10.1109/ICSM.2010.5609748.

[6] A. D. Baddeley, N. Thomson, and M. Buchanan, "*Word length and the structure of short-term memory*". *J. Verbal Learning & Verbal Behavior* **14(6)**, pp. 575–589, Dec 1975, DOI: 10.1016/S0022-5371(75)80045-4.

[7] M. Baroni, "*Distributions in text*". In *Corpus Linguistics: An International Handbook*, A. Lüdeling and M. Kytö (eds.), chap. 39, pp. 803–821, Mouton de Gruyter, 2009.

[8] G. Beniamini, S. Gingichashvili, A. Klein Orbach, and D. G. Feitelson, "*Meaningful identifier names: The case of single-letter variables*". In 25th *Intl. Conf. Program Comprehension*, pp. 45–54, May 2017, DOI: 10.1109/ICPC.2017.18.

[9] A. Berton, P. Fetter, and P. Regel-Breitzmann, "*Compound words in large-vocabulary German speech recognition systems*". In 4th *Intl. Conf. Spoken Language Processing*, vol. 2, pp. 1165–1168, Oct 1996, DOI: 10.1109/ICSLP.1996.607814.

[10] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "*To CamelCase or under_score*". In 17th *Intl. Conf. Program Comprehension*, pp. 158–167, May 2009, DOI: 10.1109/ICPC.2009.5090039.

[11] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "*Identifier length and limited programmer memory*". *Sci. Comput. Programming* **74(7)**, pp. 430–445, May 2009, DOI: 10.1016/j.scico.2009.02.006.

[12] S. Butler, M. Wermelinger, and Y. Yu, "*A survey of the forms of Java reference names*". In 23rd *Intl. Conf. Program Comprehension*, pp. 196–206, May 2015, DOI: 10.1109/ICPC.2015.30.

[13] E. Cambria and B. White, "*Jumping NLP curves: A review of natural language processing research*". *IEEE Computational Intelligence Mag.* **9(2)**, pp. 48–57, May 2014, DOI: 10.1109/MCI.2014.2307227.

[14] N. R. Carvalho, J. J. Almeida, P. R. Henriques, and M. J. Varanda, "*From source code identifiers to natural language terms*". *J. Syst. & Softw.* **100**, pp. 117–128, Feb 2015, DOI: 10.1016/j.jss.2014.10.013.

[15] C. Casalnuovo, K. Sagae, and P. Devanbu, "*Studying the difference between natural and programming language corpora*". *Empirical Softw. Eng.* **24(4)**, pp. 1823–1868, Aug 2019, DOI: 10.1007/s10664-018-9669-7.

[16] F. Deissenboeck and M. Pizka, "*Concise and consistent naming*". *Softw. Quality J.* **14(3)**, pp. 261–282, Sep 2006, DOI: 10.1007/s11219-006-9219-1.

[17] M. Dorin and S. Montenegro, "*Linguistic economy applied to programming language identifiers*". *J. Softw. Eng. & Apps.* **14(1)**, pp. 1–10, Jan 2021, DOI: 10.4236/jsea.2021.141001.

[18] S. R. Ellis and R. J. Hitchcock, "*The emergence of Zipf's law: Spontaneous encoding optimization by users of a command language*". *IEEE Trans. Syst., Man, & Cybernetics* **SMC-16(3)**, pp. 423–427, May/Jun 1986, DOI: 10.1109/TSMC.1986.4308973.

[19] A. Etgar, R. Friedman, S. Haiman, D. Perez, and D. G. Feitelson, "*The effect of information content and length on name recollection*". In 30th *Intl. Conf. Program Comprehension*, pp. 141–151, May 2022, DOI: 10.1145/3524610.3529159.

[20] D. G. Feitelson, A. Mizrahi, N. Noy, A. Ben Shabat, O. Eliyahu, and R. Sheffer, "*How developers choose names*". *IEEE Trans. Softw. Eng.* **48(1)**, pp. 37–52, Jan 2022, DOI: 10.1109/TSE.2020.2976920.

[21] R. Ferrer i Cancho and R. V. Solé, "*Zipf's law and random texts*". *Advances in Complex Systems* **5(1)**, pp. 1–6, Mar 2002, DOI: 10.1142/S0219525902000468.

[22] N. Gorla, A. C. Benander, and B. A. Benander, "*Debugging effort estimation using software metrics*". *IEEE Trans. Softw. Eng.* **16(2)**, pp. 223–231, Feb 1990, DOI: 10.1109/32.44385.

[23] R. Gresta, V. Durelli, and E. Cirilo, "*Naming practices in Java projects: An empirical study*". In XXth *Brazilian Symp. Softw. Quality*, art. 10, Nov 2021, DOI: 10.1145/3493244.3493258.

[24] M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.

[25] V. Henrich and E. Hinrichs, "*Determining immediate constituents of compounds in GermaNet*". In *Recent Advances in Natural Language Processing*, pp. 420–426, Sep 2011.

[26] J. Hieble, "*Compound words in German*". *The German Quarterly* **30(3)**, pp. 187–190, May 1957, DOI: 10.2307/401476.

[27] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "*On the naturalness of software*". *Comm. ACM* **59(5)**, pp. 122–131, May 2016, DOI: 10.1145/2902362.

[28] G. J. Holzmann, "*Code clarity*". *IEEE Softw.* **33(2)**, pp. 22–25, Mar/Apr 2016, DOI: 10.1109/MS.2016.44.

[29] D. A. Huffman, "*A method for the construction of minimum-redundancy codes*". *Proc. I.R.E.* **40(9)**, pp. 1098–1101, Sep 1952, DOI: 10.1109/JRPROC.1952.273898.

[30] A. W. Inhoff, R. Radach, and D. Heller, "*Complex compounds in German: Interword spaces facilitate segmentation but hinder assignment of meaning*". *J. Memory & Language* **42(1)**, pp. 23–50, Jan 2000, DOI: 10.1006/jmla.1999.2666.

[31] A. W. Inhoff, M. S. Starr, M. Solomon, and L. Placke, "*Eye movement during the reading of compound words and the influence of lexeme meaning*". *Memory & Cognition* **36(3)**, pp. 675–687, Apr 2008, DOI: 10.3758/MC.36.3.675.

[32] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "*Big code != big vocabulary: Open-vocabulary models for source code*". In 42nd *Intl. Conf. Softw. Eng.*, pp. 1073–1085, Oct 2020, DOI: 10.1145/3377811.3380342.

[33] W. Li, "*Random texts exhibit Zipf's-law-like word frequency distribution*". *IEEE Trans. Inf. Theory* **38(6)**, pp. 1842–1845, Nov 1992, DOI: 10.1109/18.165464.

[34] W. Li, "*Zipf's law everywhere*". *Glottometrics* **5**, pp. 14–21, 2002.

[35] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009.

[36] G. A. Miller, "*Some effects of intermittent silence*". *Am. J. Psychology* **70(2)**, pp. 311–314, Jun 1957, DOI: 10.2307/1419346.

[37] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "*Curating GitHub for engineered software projects*". *Empirical Softw. Eng.* **22(6)**, pp. 3219–3253, Dec 2017, DOI: 10.1007/s10664-017-9512-6.

[38] M. Perc, "*Evolution of the most common English words and phrases over the centuries*". *J. Royal Soc. Interface* **9(77)**, pp. 3323–3328, Dec 2012, DOI: 10.1098/rsif.2012.0491.

[39] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "*An empirical investigation of how and why developers rename idnetifiers*". In 2nd *Intl. Workshop on Refactoring*, pp. 26–33, Sep 2018, DOI: 10.1145/3242163.3242169.

[40] S. T. Piantadosi, "*Zipf's word frequency law in natural language: A critical review and future directions*". *Psychonomic Bulletin & Review* **21(5)**, pp. 1112–1130, Oct 2014, DOI: 10.3758/s13423-014-0585-6.

[41] S. T. Piantadosi, H. Tily, and E. Gibson, "*Word lengths are optimized for efficient communication*". *Proc. Natl. Acad. Sci. U.S.A.* **108(9)**, pp. 3526–3529, Mar 2011, DOI: 10.1073/pnas.1012551108.

[42] M. Popović, D. Stein, and H. Ney, "*Statistical machine translation of German compound words*". In *Advances in Natural Language Processing*, T. Salakoski, F. Ginter, S. Pyysalo, and T. Pahikkala (eds.), pp. 616–624, Springer, Aug 2006, DOI: 10.1007/11816508_61. (LNCS vol. 4139).

[43] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "*Descriptive compound identifier names improve source code comprehension*". In 26th *Intl. Conf. Program Comprehension*, pp. 31–40, May 2018, DOI: 10.1145/3196321.3196332.

[44] B. Sharif and J. I. Maletic, "*An eye tracking study on camelCase and under_score identifier styles*". In 18th *Intl. Conf. Program Comprehension*, pp. 196–205, Jun 2010, DOI: 10.1109/ICPC.2010.41.

[45] O. Tripp and D. G. Feitelson, *Zipf's Law Revisited*. Tech. Rep. 2007-115, Hebrew University, Aug 2007.

[46] H. Zhang, "*Exploring regularity in source code: Software science and Zipf's law*". In 15th *Working Conf. Reverse Eng.*, pp. 101–110, Oct 2008, DOI: 10.1109/WCRE.2008.37.

[47] G. K. Zipf, *The Psycho-Biology of Language: An Introduction to Dynamic Philology*. Houghton Mifflin, 1935.

[48] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.