Software Identifier Naming Conventions & Dictionary

DOV BENYOMIN SOHACHESKI
Department of Software Engineering
SCE - Shamoon College of Engineering
84 Jabotinsky St. Ashdod, 77245 Israel
ISRAEL

YOTAM LURIE
Department of Management
Ben-Gurion University of the Negev
P.O.Box 653, Beer-Sheva, Israel 84105
ISRAEL

SHLOMO MARK
Department of Software Engineering
SCE - Shamoon College of Engineering
84 Jabotinsky St. Ashdod, 77245 Israel
ISRAEL

Abstract: - Software developers have been presented with so many tools meant to assist then during the development process. Tools like autocomplete, intelli-sense, linters, and other static analysis solutions. All such tools have one underlying goal, to promote productivity and improve quality. Much research has been conducted on the topic of software quality and its direct benefits both during and after the development cycle. Various methods of measuring and improving quality in software products have been implemented at a grand scale. However, software developers are still left with the choice of implementation details. One such detail is the choice of identifier names in the code written. Few publications have focused on conventions, guides, or best-practices on the topic of identifiers naming choices (not to be confused with coding styles). Much time and energy is misused by developers while choosing an appropriate identifier name, as well as by other developers later on when trying to understand the choice made by their colleagues. By aggregating and compiling a list of readily available identifier names that developers can choose from, will allow them to focus on other keys aspects of development.

Key-Words: - Quality, Identifiers, Styles, Conventions, Best-practices.

Received: January 30, 2021. Revised: June 28, 2021. Accepted: June 30, 2021. Published: July 6, 2021.

1 Introduction

Philosophy debates whether humankind fundamentally obeys a predefined set of universally accepted truths. For example, we are contracted by the confines of the laws of physics and nature. The limit of these confines is credited for creating the bounds of our options, choices, and decisions. The accuracy of such a proposition is subject to rebuttal [1] because the definition encompasses a broad scope of all of humankind and its absoluteness. Nonetheless, these propositions can be partially accepted, in the sense that we do indeed bear engraved traits or instincts that regulate our habits and demeanor. Our customs and conduct are consequences of an intrinsic natural process that is not governed by human made law [2]. Therefore, it is natural and even expected for a group of people to absorb a given concept with distinctively contrasting perceptions.

Software development can be conducted by any person willing to adhere to a set of predefined rules and procedures for a given software language. Software engineering on the other hand, is a science that invests in the human-factor along-side the

mundane development of code. Even during its infancy, software engineering introduced many new methods of thought and development schemes that were also popularized and implemented in other scopes of science. Since its advent, the various methods have sustained scrutiny and reform at a rate never before seen in other fields of engineering. Methodologies and practices, previously accepted and even deemed preferable, have grown outdated and impractical [3]. The evolution of the methods was a by-product of various obstacles. One fundamental deficiency was the absence of a profound understanding of the human factor and its influence on the software process. Agile, a contemporary and widely implemented methodology, attempted to repair the shortcomings of other methods by headlining the human factor and ergonomics [3]. Agile principles lead a project throughout its various lifecycle approaches and even continues with post-completion guidance. Agile proposes numerous conventions and patterns concerning the actual development process [4] [5] [6]. It also channels on the mechanisms to boost better communication between the various stakeholders. Factoring all its interpersonal contributions and methodological enhancements, nevertheless, agile does not impose any provisions on the implementation of a project's workflow. Therefore, coding styles and conventions remain an organizational choice.

Among the programming paradigms, there are three major classifications: functional, object-oriented, and procedural. Paradigms are usually connected to a programming language's execution model and the sequence in which operations are invoked [7]. Still, a paradigm does not impact the styles or conventions used in that programming language. One of the first works, outlined by Brian W. Kernighan, presents the notion of how to structure code, obtain user input and output, and scaffold project architectures [8]. Although Kernighan's code snippets are nearly 40 years old, we can extract his principle argument that programming styles assist in making code easier to understand and consume. Many new programming languages have been introduced since its publication and yet, the underlying teachings are still timeless and valid.

All programming languages have individual "flavors" and conventions. An organization's decision to utilize a given coding flavor is a choice that lands under the auspices of software quality. In order to preserve a fluid style within organizations, it is customary to adopt commonly known style guides distributed by leading actors in the software development field. A style guide is a list of recommendations and practices proven for enhancing program comprehension and readability. Software programs are a series of predefined keywords, syntactical structures, and the naming of variables, functions, methods, and classes known as identifiers. Since developers are free to choose names for identifiers in their code, it is crucial that guidelines and conventions are followed to guarantee consistence throughout the codebase. Agreeing on a certain style guide is more than just a preference, it allows for the extraction of quality metrics [9]. "Code smells", addressed in the upcoming section, is a contemporary metric scale for software quality [10] [11]. The worldwide demand and consumption of technology is perpetually growing; technological changes are frequent and aggressive. Software engineering is therefore compelled to remain fluid in its supplying solutions at such an accelerated rate. Like many other evolving fields, software engineering is still ratifying many of its core concepts and principles. Many definitions of principles and approaches have yet to be finalized in academic literature or applied in the software industry. While other fundamentals are still subject to debate. One substantial theme in software engineering that has yet to detail its definitions unanimously, is the methods and scales of measurements, as they pertain to software quality. The topic of software quality has gained much popularity over the years; as such, many new methodologies and approaches have placed quality at the epicentre of their proposed workflows. With the increase of quality measure and tooling, projects can drastically raise the overall level of the product quality. The above-mentioned concern can be resolved by the establishing of identifier dictionary discussed in upcoming sections of this paper.

2 Scientific Background

The following scientific background includes an overview of the relevant theoretical and practical concepts for the research. First, a brief summary of the current state of code conventions and available tooling will be presented. Subsequently, we will explore the correlation between software assessment tools and software quality.

2.1 Linters

The word lint comes from English and refers to the "fuzz consisting especially of fine raveling and short fibers of yarn and fabric" [12] that form small balls that dangle from garments. "Lint" was first introduced to software engineering by Stephen C. Johnson of Bell Labs (a prime contributor to UNIX) [13], he projected the English understanding of the word lint to the software realm by developing a tool named "lint". The tool analysed *C* code and in Johnson's words [14] "enforces the typing rules of *C* more strictly than the *C* compilers". Today, many automated scripts have been authored for the majority of programming languages that follow the same original principles as Johnson's lint, they are commonly known as "linters". They are commissioned to statically analyse code and emit warnings for deviations from style guides [9], or common traps that usually evolve into bugs. A linter analyzes code based on a predefined collection of rules and preferences. Linters come predefined with a set of essential rules that satisfy a large scope of software code. It is important to note that the linting process is meant to assess code quality and is not a related to specific programing paradigm. Pylint used to lint *Python*, and *ESLint* used for Node.js or *Javascript* are two well-recognized linters.

Projects that utilize linters can overwrite or extend any rule to conform to their organizational style of coding. Research has determined that roughly half of open-source projects implement linting as part of their project workflow. However, the majority of those projects rely on the prebaked set of rules and do not override or extend the defaults [14]. Other research has examined the cognitive inconsistency in open-source projects and its effects on attracting new contributors to the project [15]. Meaning, the absence of a uniform style and flow within the code base, leaves contributors feeling disoriented, not knowing which style or form to use for their contribution.

Linters are capable of identifying and reporting several groups of infractions listed briefly below in no order of significance or severity [9]

1. *"Errors":* normally signify concrete flaws that will propagate during compilation. For example, the invoking of an undefined method or other language-syntactic errors.
2. *"Warnings":* are less severe than errors and in some cases will never lead to an actual error. They can be classified as not complying to best-practices. For example, logging information to the console, which can be considered inappropriate in production environments.
3. *"Code smells":* refer to code that can be restructured or refactored to improve the readability and maintainability of the code. This category will be discussed further in an upcoming section. An example of a code smell can be the depth complexity of function or the duplication of code blocks.
4. *"Conventions"*: are guidelines that concentrate on structural quality and fluidity of code. For example, the location of the placement of curly braces (i.e. {}), the maximum amount of blank lines between methods or blocks, as well as the naming of files, methods, and variables.

2.2 Style Guides

One of the foundational traits of linters is to drive developers to adhere to a defined style best-practice. As mentioned above, conventions are a collection of practices that suggest a certain style or practice for coding. Many organizations publish styles guides in the form of best practices for coding readability. Google provides style guides for C++, Java, Python, and nearly a dozen additional languages. Their goal is clearly asserted as being, "much easier to understand a large codebase when all the code in it is in a consistent style" [16]. Other companies, like Airbnb, are more opinionated in their claims that their style guides are, "a mostly reasonable approach to JavaScript" [17]. ESLint offers the possibility to enable recommended rulesets based on popular style guides, such as that of Airbnb. The principal objective of guides and best-practices is to separate arbitrary decisions making from the developer and make the codebase more maintainable and readable. In essence, several developers can commit code to a common repository and lend the impression the code was authored by a single developer.

2.3 Code Quality & Maintainability

There are many opinions related to the definition of quality in software engineering. The following section presents the approach adopted during our conducted research. The word "quality" is generally understood to be, "degree of excellence" or a "distinguishing attribute" [18]**.** In other words, quality is the measure of a positive value that an item or concepts embodies. The perception of "code quality" can be assessed on an adversity basis by software developers. Some developers prefer concise code and regularly refactor their code to obtain pure and sparse code blocks. While other developers target verbose code that is self-explanatory [19]**.** Both developer types consider their approach to be of higher quality and that of their peer to be feeble and lacking in quality. J.M Juran provides two

approaches to the definition of code quality in software engineering:

1. Quality is a measure of product features which meet the customer's requirements and thereby provide product satisfaction.
2. Quality means the freedom from deficiencies [20]

The first definition simply relates to accomplishing the customer's requests and can relate to quality as more of a business value than a software value [21]**.** The second definition relates to quality as a state of the product. Meaning, the product has an absence of errors that would in turn require the development team to repair or maintain the code in order to repair them. It is important to note that both definitions do not relate to the actual coding styles or conventions.

2.4 Code Smells

Refactoring is the process of enhancing code quality without appending new features [11] Software systems are not shielded from discord, on the contrary, due to their complexity they are more prone to disarray and chaos than other physical products. During the lifetime of a software system, it will be subject to constant revision and as well as extensions. As such, the overall quality of the codebase begins to decline, necessitating the use of refactoring.

Some of the warnings emitted by linters advise of a possible refactor in the future. This measure is called a *bad code smell* [9]. Moreover, code smells are segments of code, which, under the current state of the system, can achieve an improved quality rating. These sections are also common pitfalls for developers when maintenance is required [10]. Beck and Fowler introduced 22 code smells and their correlated strategies for refactoring and eliminating the smell [11] [10].

3. Research Objectives and Expected Significance.

The human factor directly impacts the process' quality [22] [3]. Teamwork and collaboration are at the core of development lifecycle. Communication, interaction, and understanding among team members are continuous and central to the success of any project. The human factor of communication that can benefits a project is the ability of team members to clearly express their perspectives both verbally and non-verbally. Research has been conducted on the performance improvements amongst teams whose members yield cognitive similarities [23]. The following section assumes that a team project was developed containing no code-smells and complied to a style recommendation. When members of an organization or team come in contact with program code authored by others, they are faced with the challenge of understanding its purpose. Even the most experienced developers are forced to apply the mindset of the programmer who originally drafted the code to comprehensively discern their intentions [9]. In many cases, the code is intrinsically complicated (and not due to its logic or algorithm), as such, would pose a difficulty for the original author themselves. In cases where software quality and code smells are fundamental concerns that are baked into the software process, there may still exist a perception or cognitive par that must be bridged in order to understand the code.

Software code consists of predefined keywords and syntactical structures that are language specific and as well as the naming of variables, functions, methods, and classes known as identifiers. Developers are granted the freedom to choose names for identifiers in their code. Such identifiers can be categorized in one of two ways:

1. Obscure: Identifier names that are not dictionary terms or spoken words and introduce ambiguity to the codebase. This family can be further partitioned:
2. Under certain circumstances, a developer can deduce the meaning of the identifiers only with when assisted by the context in which it was used. For example, using the variable name e in the context of error-handling or i in a looping structure have no meaning when presented outside their context.
3. Other names have no innate meaning whatsoever and are completely arbitrary. Even when accompanied by their code context they do not contribute a deeper understanding behind the naming choice or the intent of the code itself. For example, variables named a, b, c or n1, n2, and tmp.

2. **Implicit:** For the most part, there are no rights or wrongs when determining identifiers. However, the developer should possess the desire to portrait the unit under development, therefore, the names chosen will usually directly relate to their personal understanding of the feature being developed and the context in which it exists. This family can be further subdivided into two categories:

1. The identifier names preferred by the developer would match that of the majority of other developers tasked with coding the same unit.
2. The identifier can be understood without delving into the context of the unit being developed, but would not likely be a first choice for the majority of other developers. Our perception of the domain and context may vastly differ from a that of a

colleague developing alongside us. These deviations introduce new cognitive complexities to the comprehension of software code. For example, the procedure of saving information into a database can be named store, save, persist, or update. The first three options imply retention for future use, whereas update can be considered equivocal and only fully understood given the context of its use.

This research pertains to a subset of software development organizations. Members of the subset are assumed to be:

1. Organizations that implement and conforms to predefined coding convention and style guide.
2. The organizations also stress the importance of software quality during the development process.

These subsets were chosen in order to refine our results sets and sift away intrinsically poor-quality code. This can be compared to someone seeking to learn a foreign language would prefer to learn from a native speaker, guaranteeing quality results. With these assumptions in mind, the following section will present the research objectives. The overall aim of this research is to assist software developers throughout the development process in improving the language infused into the codebase. Meaning, presenting the developer with a predefined dictionary of terms and their association to a given context.

In conclusion, this research project has seven aims:

Aim 1: Define the membership characteristics for the obscure and implicit identifier categories.

Aim 2: Textually mine identifiers in software code to forge a preliminary dictionary of terms.

Aim 3: Forge a final dictionary by classifying terms into the above-mentioned categories.

Aim 4: Suggest implicit identifier names and warn upon the use of obscure names to developers during the development process.

Aim 5: Inspect the impediments and constraints in applying such concepts in practice.

Aim 6: Measure the cognitive benefits and comprehension of code when using naturally understood terms.

Aim 7: Determine the boundaries of Natural Language Process (NLP) [24] processing as it pertains to analyzing a codebase for departures from accepted naming conventions.

4. Detailed Description of the Proposed Research.

4.1 Research Methods

The proposed research proceeds from the assumption that software quality can be improved when implicit naming conventions are chosen over obscure ones. Moreover, identifier names with a higher rank present an even greater improvement in quality than that of less commonly used names.

4.2 Methods

The overall research project employs a mixed research method, with elements of both quantitative and qualitative research [24]. Data will be collected from by means of text mining and textual analysis of readily available online software projects. This approach requires the development of a software tool to assist in the gathering and extracting of relevant data.

The project aims to provide a strong, empirically based assessment of the current state of software styles and conventions. We will also attempt to identify the circumstances and consequences for deviating or disregarding conventions altogether. That is, all elements of the study bestow the same general theme and apply the same data samples, yet have different scopes, variables, and designs.

4.3 Sampling Strategy

We recognize a software developer as a person concerned with all aspects of the software development process. That includes, but is not limited to, people conducting research, analysis, design, programming, testing, and management activities in the field of software development. It also encompasses the development of software for the purpose of work, as a hobby, or simply from passion. Obtaining sample data that would generalize the software developers' population is a challenge; since we cannot accurately distinguish the number of software developers worldwide, nor have the means of reaching them. Following several previous studies [25], we will rely on the online social community known as GitHub (github.com) to capture pertinent samples for our research.

GitHub is a software development platform that enables hosting of software assets, code reviewing, project management, open collaboration, and more. The GitHub community is considered a social coding community, the second in popularity to Facebook, with more than 800 million registered users of which 320 million are active monthly [26]. The design of the project, as mentioned above, consists of data analysis of openly available source code (from

GitHub) in order to classify the naming conventions adopted in software repositories. The classification will be converted into a dictionary of terms, itemized by popularity and assumed degrees of understanding. The actual methods used to extract data from the online packages is described in the ensuing section.

5 Research Design

5.1 Data Analysis Design

We chose to use GitHub as a source for the data we will study during the research process due to its availability and wealth of options. GitHub is non-discriminant in the languages it supports, however, Javascript is unquestionably the most popular language on the platform [27]. Furthermore, a fundamental part of the Javascript ecosystem, like Node Package Manager (NPM), intrinsically encourage open-collaboration. Codebases that welcome collaboration habitually contain code that was authored by developers from ranging educational backgrounds and demographic diversity. It is plausible that the sole similarity connecting two collaborators is the project they contribute to. The diversity among collaborators will be to our advantage, as it will afford a rich and extensive spectrum of identifier name for our dictionary.

Website user experience can be accomplished using Javascript. As such, web-designers, who usually focus on design aspects and not formal engineering tasks [28], attribute a large portion of the implementors of Javascript. However, we will omit browser-based Javascript for a reason addressed in the subsequent paragraph as it regards to Python.

A second language that we considered to analyze was Python. We elected not to analyze Python programs for the following key reason. Python is regularly used as a scripting language by scientists, IT, and even programmers to accomplish single-responsibility tasks [29]. These tasks are contained in single files that maintain all their dependencies. Therefore, architecture, styles, and conventions are not provided precedence.

5.1 Data Analysis Tool

We designed and developed a software tool to assist in the necessary extraction of statistical data from online software projects. The tool was developed using the Node.js programming language as a base, with compulsory modules written in Python. The tool is packaged with an intuitive user-interface, allowing any locally-stored project to be analyzed. There are two main branches of the tool: the analysis pipeline and the result reports.

The analysis pipeline is composed of several non-concurrent, consecutive steps that resolve a software project into its key components. Once a project root directory is selected, the following steps are executed:

1. **Load Project Files:** starting at the project's root, all files are recursively read into memory. For performance reasons, only the path to the file and its related meta-data related to the file are loaded into memory and not the content of the file itself. We decided to ignore the actual content of the files at this stage in the pipeline for performance reasons; the file may be completely ignored in the subsequent step of the pipeline.
2. **Ignore Unnecessary Files:** a file is considered unnecessary and will be ignored from the project analysis if does not meet a set of predefined criteria. The criteria we established are based on our assumptions presented in the section on sampling strategies. The list of criteria is as follows:
    1. **Only JavaScript:** the file must be a JavaScript file. Therefore, we ignore all files that do not have the extension js. We recognize the fact that file extensions do not guarantee the actual content type of the file. We also recognize that modern JavaScript can be written using other extensions, in which case a transpiler is utilized to convert the code to native JavaScript. Nonetheless, when applying a more rigorous filter we can be assured that our results do not contain irrelevant code.
    2. **Development Directory:** the file must be located in a directory that contains development code. Similar to the previously mentioned rule, this filter is rather strict in its refinements. All files found in a distribution, testing, or configuration directory will be ignored. This allows us to focus our analysis on code that was intended for development and not the minified or compiled (per-se) code intended for the end user.
    3. **Non-Hidden Files:** all hidden files and folders are removed from analysis. This usually includes configuration files or Git repository history details.
3. **Extract Identifiers:** at this point in the pipeline, all files are considered pristine. The content of the files are loaded into memory in an asynchronous manner. Once all files have been loaded, they are iterated over and transformed into a collection of tokens, while filtering language specific keywords and constructs.

Each token is in essence an identifier name and the location in the file (line and column) in which the identifier was found.

4. **Run NLP:** using the collection of identifiers from the previous pipe, we attempt to build language context data for each identifier. The context attempts to split the identifier into separate words. For example, for the identifier setDomainResolutionProtocol, the words set, domain, resolution, and protocol will be extracted. The extraction can parse identifiers written using the four common styles: camelCase, PascalCase, kabob-case, and snake_case.

We implemented spaCy in our project as a third-party NLP engine because it is open-source and already collection of comprehensive, pertained models. Once the words have been separated, they are analyzed as a sentence by the NLP engine. I.e., the identifier setDomainResolutionProtocol will be read as "set domain resolution protocol". The NLP engine responds to the sentence query with a mapped annotation and lemma for each word in the sentence. A lemma is a "reduced inflectional forms and sometimes derivationally related forms of a word to a common base form" [30]. For example, when "set sorting algorithm" is queries, the word "sorting" is reduced to the lemma "sort", with a verb annotation.
The last portion of the context processing is checking the rating or frequency of the queried word. Rating and frequency are based on Google's N-Gram top ten-thousand words [31]. The rating will allow us to determine the differences between a spoken-language and programming-language. When a word does not exist in the list of 10,000 words, a broader, more encompassing list of 475,000 english words is queried. If the word is not included in the larger list, it is defined as misspelled (obscure).

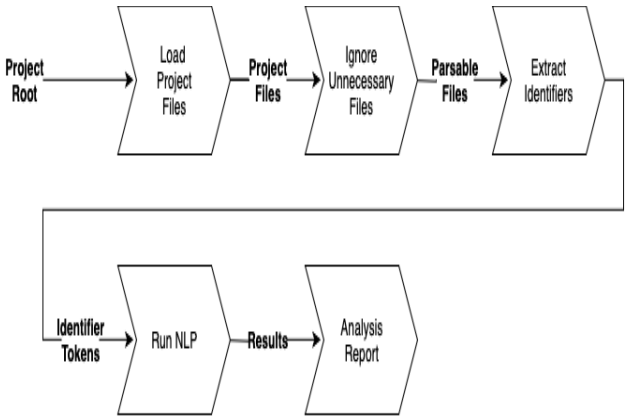

Figure 1. Data Analysis Pipeline

Following the pipeline, the result reports is presented, which contains statistical conclusions. The structure and details of the report are presented in a subsequent section.

6 Data Analysis

Our data analysis tool was designed to scan a software project repository and generate a result set or report. Before we present our results, we must first define a series of entities, sets, and functions that are crucial to the proper understanding of the results.

6.1 Entities

- **Identifier:** any raw variable, function, method, class, parameter, or property name found in a software repository. In other words, any word that is not a predefined keyword for the syntax that language.
- **Lemma:** defined above as, "reduced inflectional forms and sometimes derivationally related forms of a word to a common base form".

6.2 Sets

- Identifiers: a multi-set or bag of all the identifier derived from the scanned project. The multi-set is denoted by the letter I.
- Lemmas: a multi-set or bag of all lemmas derived from the scanned project. The multi-set is denoted by the letter L. In order to compute certain calculations, we need to transform the multi-set L into a set with unique elements. This unique set is denoted by L'.

$$L' = \bigcup_{\forall l \in L} \{ l \}$$

- **English dictionary:** a set of all word found in the english language, in all tenses and pluralizations. The set is denoted by the letter E.
- **Common English dictionary:** a set of the 10,000 most popular words in the english language. The set is denoted by E'. It is important to note that $E' \subseteq E$.

Functions

- **Frequency:** the total amount of times a lemma was found in any identifier name. The count does not have to refer to a unique instance of the lemma. Meaning, if an identifier was used more than once, all of its lemmas' counts will be incremented. The function is neither surjectived or injective and is denoted by (where N is the set of natural numbers):

$$freq : L \rightarrow \mathbb{N}_{\geqslant 1}$$

- **Rank:** defines the how common a lemma is based on its frequency in the project. The most common lemma will receive a rank of 1 followed by

the second most common with a rank of 2 and so on; the higher the frequency the lower the rank. If more than one lemma has an equal frequency, then the lemmas will be provided sequential rank based on alphabetic order. The function is both surjectived or injective and is denoted by:

$$rank : L \rightarrow [1, 2, 3, \ \ldots, |L'|]$$

- **English rank:** similar to the previous function, however the rank is based on a predefined list provide by Google and is based on how common a word is in the english language. The function is denoted by:

$$eng : E' \rightarrow [1, 2, 3, \ \ldots, |E'|]$$

- **Distance:** the difference between the rank of a lemma in the english language versus its rank in the scope of a software project. This function is very important because it expresses the contrast between NLP used in the english language and a similar model (that has yet to be defined) for the software language. For example, the word "the" is considered the most popular english word (i.e. eng(the) = 1) however, the word "the" is almost never found as an identifier name in software projects (i.e. rank(the) = ∞). The function is denoted by:

$$dist : L \rightarrow \mathbb{N}_{\geqslant 1}$$
$$dist(l) = \begin{cases} rank(l) - eng(l) & l \in \mathbf{E}' \\ \infty & l \notin \mathbf{E}' \end{cases}$$

We initially defined rank(l) - eng(l) within an absolute value expression, eliminating the distinction that can be made by sign (negative or positive) of the function's result values. The following example will assist us in discerning between four alternative results:

1. **Infinity — dist(l) = ∞:** occurs when a term does not exist within the list of popular english words. Such a circumstance usually points to an identifier name that is unique to the code-base for business logic purposes. For example, the word "latency" does not exist in the list of popular english words, however can easily be found in a software program.
2. **Zero — dist(l) = 0**: occurs when the popularity of lemma in the english language is equal to the popularity within the code-base. This would imply that the NLP model used to analyze the english language could be directly applied as the software analysis model.
3. **Negative values — dist(l) < 0:** occur when a lemma is more popular within a code-base then the english language. For example, the word "set" is very popular *(usually one of the top 10 most used identifier terms)* in software code, whereas *eng(set) = 189*.
4. **Positive values — dist(l) > 0:** occur when a lemma is less popular within a code-base then the english language. For example, the word "in" is very popular in english, *eng(in) = 6* and is also used in software code, albeit less frequently.

- **Degree:** provides insight into the broad use of a lemma and not just its frequency or rank. The degree function counts the amount of unique identifiers that contain a given lemma. This is needed in order to sift through popular, yet not ideal identifier names. For example, a commonly used function with a flawed identifier name can be introduced into a software repository. Its frequency will be very high, however its degree will be very low because the flawed choice of lemmas will not be found in other identifiers. The function is neither surjectived or injective and is denoted by:

$$degree : L \rightarrow \mathbb{N}_{\geqslant 1}$$

An example for the relationship between a lemma and the amount of identifiers that contain it can be seen in the diagram below, where degree(file)=5 because five identifier contain the lemma file:

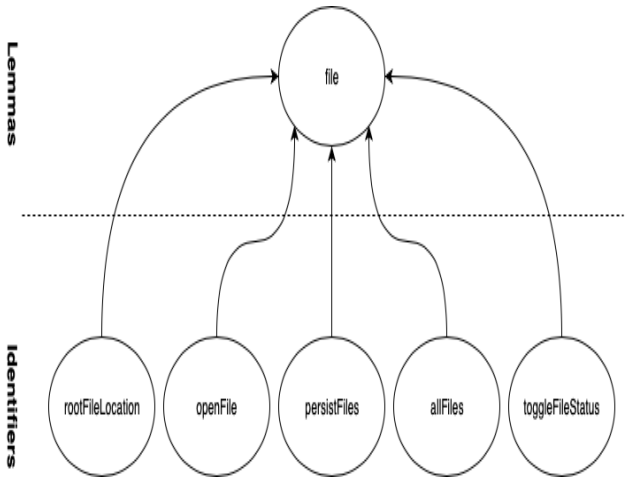

Figure 2. Identifier to Lemma Relationship

- **Related:** specifies which lemmas are used together to compile an identified name.

$$related : L \rightarrow [L_1, L_2, \ \ldots \ L_{|L|}]$$

This function is very useful when referring to a compound identifier *(compiled of two or more lemmas)* as a spoken sentence. This can be illustrated using an undirected graph: the nodes of the graph are the lemmas found in a repository and the edges between two nodes represent those nodes *(or lemmas)* being used together in the same identifier. The figure below displays a graph for the following group of identifiers: setDomainResolutionProtocol, setName, setDate, setUserName, userName, userLogin,userPassword loginName, loginPassword domainName, loadDomain, resolvePassword, fileName, loadFile, and fileProtocol.

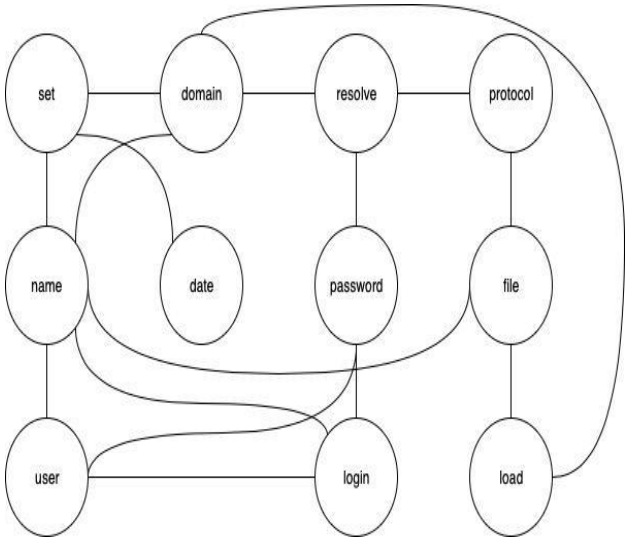

Figure 3. Lemma Relationship Graph

Using the example undirected cyclic graph above, the function *related(user) = [password, set, login, name]*. This is useful when attempting to suggest a lemma during development. Meaning, based on previously related lemmas, we can recommend the use of chained lemmas for the same identifier name. Currently, emphasis has not been placed on the direction of the vertices of the graph. For example, assuming the developer has begun writing a new identifier name and has only written user, we can now accurately propose related lemmas to them to complete their identifier name.

7 Conclusion

This research has discussed the measurement of code quality using tools like linters and measurement scales such as code-smells. The research's main focus was on attempting to establish an additional method of measuring, as well as improving, code quality. With the abundance of readily available tools to measure code quality (both open-source and proprietary), none of them provide a means for measuring the usefulness or apprehension of an identifier name within a codebase. Our attempt to create an identifier dictionary would both assist developers in choosing preferred identifier names as well as reveal code that already contains aimless names.

In order to substantiating the mechanism that would be used to analyze software repositories and establish a measure of quality pertaining to identifier names. We were able to develop a tool that is generic and impartial to the programming-language that it is analyzing. The tool was successful in recursively scanning complete code repositories and extract the identifier names used throughout. The tool we developed was also able to correctly split the identifier names in to their base "lemmas", turning a compound identifier into an understood "spoken sentence". Part of the process of understanding an identifier name as a spoken sentence is accomplished by means of an NLP engine. It became immediately evident that the NLP model used by the engine was insufficient for our goals. The spoken english language is closely related to the terminology found within software code but with expected modification. Further research would demand alterations to the underlying model used in the NLP engine.

As a third step in our research, we defined a logical and mathematical scale that should be used to sort through a batch of identifier names gathered from a software repository. The "sets" and "functions" defined were based on our preliminary tests of our extraction tool. Each set and function have a specific goal in both understanding and defining the usefulness of an identifier name in a given scope.

Our research forms a solid base for the demand of an identifier dictionary among developers, as well as the benefit it would have to software quality. Likewise, this research can be used as a stepping stone for further research. The sets, functions, and scales we defined can be used to define meta-like data used by a learning NLP engine or any other form of artificial intelligence engine. Such improvement could supply real-time suggestions to developers and improve software code while it is being coded. The level of competence at which a programmer writes software code does not directly reflect their personal views for

what identifier names are considered more appropriate and at what times. The naming suggestions are intended for software developers at every level, novice and professional alike. This newly proposed measuring scale is statically executed and does not need to be directly implemented as part of the development workflow, it can be run subsequently by during a CICD pipelines or by other automated tooling. The proposed identifier naming suggestions would evolve as part of a developer's workflow and become no different than linters and styles guides already present in so many projects.

A developer should be primarily focused on the task they are attempting to solve and not sidelined with responsibility of choosing an identifier name. With our tool in place as both a development assistant as well as a method for measuring quality after the fact, developers be able to provide paramount attention to architecture and functionally. This would automatically promote quality, productivity, and time-to-market all at the same time.

References:

[1] F. I. Dretske , "Laws of Nature," *Philosophy of Science*, vol. 44, no. 2, pp. 248-268., 1977.

[2] A. Perreau-Saussine and J. B. Murphy , The Nature Of Customary Law., Cambridge University Press., 2007.

[3] A. Cockburn and J. Highsmith, "Agile software development, the people factor.," Computer, vol. 34, no. 11, pp. 131-133., 2001.

[4] Y. Lurie and S. Mark, "Professional Ethics of Software Engineers: An Ethical Framework.," Science and engineering ethics, vol. 22, no. 2, pp. 417-434., 2016.

[5] S. Mark and Y. Lurie, "Customized Project Charter for Computational Scientific Software Products.," Journal of Computational Methods in Sciences and Engineering, vol. 18, no. 1, pp. 165-176., 2018.

[6] H. Abdulhalim, Y. Lurie and S. Mark, Ethics as a Quality Driver in Agile Software Projects. Journal of Service Science and Management, 11(01), 13-25., 2018.

[7] P. Van Roy, "Programming paradigms for dummies: What every programmer should know," New computational paradigms for computer music, vol. 104, pp. 616-621., 2009.

[8] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, 2nd ed., McGraw Hill., 1978.

[9] S. Boutnaru and A. Hershkovitz, "Software quality and security in teachers' and students' codes when learning a new programming language," Interdisciplinary Journal of e-Skills and Life Long Learning, vol. 11, pp. 123-147., 2015.

[10] A. Yamashita and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection?–an empirical study.," Information and Software Technology, vol. 55, no. 12, p. 2223–2242., 2013.

[11] M. Mäntylä, J. Vanhanen and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code.," in The 19th International Conference on Software Maintenance (ICSM 2003), 2003.

[12] Merriam-Webster, Lint, Merriam-Webster.com dictionary., 2020.

[13] S. C. Johnson and E. L. Michael, "UNIX time-sharing system: Language development tools," The Bell System Technical Journal,, vol. 57, no. 6, pp. 2155-2175., 1978.

[14] S. C. Johnson, Lint, a C program checker., Bell Telephone Laboratories., 1977.

[15] M. Beller, R. Bholanath, S. McIntosh and A. Zaidman, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software.," in The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2016.

[16] Z. Wang and J. Hahn, "The Effects of Programming Style on Open Source Collaboration.," in ICIS 2017 , 2017.

[17] GitHub, Google/styleguide., 2018.

[18] GitHub, Airbnb/javascript., 2018.

[19] M.-W. dictionary, Quality., 2020.

[20] Smartbear, Defining Code Quality., Smartbear.Com., 2015.

[21] M. J. Juran and A. Blanton Godfrey, Quality handbook, 5th ed., McGraw-Hill., 1999, pp. 173-178.
[22] R. J. Williams and E. Dietrich, What Does Code Quality Actually Mean? Dzone Agile., 2017.
[23] A. Chagas, M. Santos and A. Vasconcelos, "The impact of human factors on agile projects.," in Agile Conference (AGILE), 2015.
[24] H. R. Kang, H. D. Yang and C. Rowley, "Factors in team effectiveness: Cognitive and demographic similarities of software development team members.," Human Relations, vol. 59, no. 12, p. 1681–1710., 2006.
[25] H. Cunningham, "A definition and short history of Language Engineering.," Natural Language Engineering, vol. 5, no. 1, pp. 1-16., 1999.
[26] J. W. Creswell, "Mapping the field of mixed methods research.," Journal of of mixed methods research, vol. 3, no. 2, pp. 95-108., 2009.
[27] G. Gousios, M. A. Storey and A. Bacchelli, "Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective.," in 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion, ICSE 2016, 2016.
[28] A. Giri, A. Ravikumar, S. Mote and R. Bharadwaj, "Vritthi-a theoretical framework for IT recruitment based on machine learning techniques applied over Twitter, LinkedIn, SPOJ and GitHub profiles.," in Data Mining and Advanced Computing (SAPIENCE), 2016.
[29] H. Borges, M. T. Valente, A. Hora and J. Coelho, "On the popularity of GitHub applications: A preliminary note.," 2015.
[30] T. Mikkonen and A. Taivalsaari, "Using JavaScript as a real programming language [Report].," Sun Microsystems., 2007.
[31] F. Perez, B. E. Granger and J. D. Hunter, "Python: an ecosystem for scientific computing.," Computing in Science & Engineering, vol. 13, no. 2, pp. 13-21., 2010.
[32] C. D. Manning, P. Raghavan and H. Schütze, "Stemming and lemmatization. Introduction to information retrieval," 2008.
[33] I. Google, "Google Ngram Viewer," 2019.

Creative Commons Attribution License 4.0 (Attribution 4.0 International, CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0
https://creativecommons.org/licenses/by/4.0/deed.en_US