Comparing Identifiers and Comments in Engineered and Non-Engineered Code: A Large-Scale Empirical Study

Otávio Lemos, Marcelo Suzuki, Adriano de Paula
{otavio.lemos, marcelo.suzuki, adriano.paula}@unifesp.br
Science and Technology Dept.
Federal University of S. Paulo
S. J. dos Campos, SP, Brazil

Claire Le Goes
clgoes@cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA

ABSTRACT

Identifiers and comments are among the most important sources of information in software. Indeed, an obvious way to understand a piece of code is to read the available comments. When these are not present, program comprehension depends strongly on informative identifier names. In this paper, we investigate the relationship between the use of solid software engineering (SE) practices — such as unit testing and continuous integration — and features of identifiers and comments. To gather code developed with and without the use of well-established SE practices, we adopted a previously developed classifier that predicts whether a repository on GitHub contains an *engineered* software project. We collected approximately 6,000,000 identifiers — class, interface, field, method, and variable names — as well as 6,000,000 comments from Java code classified as either engineered or non-engineered. We find that identifiers and comments differ significantly between engineered and non-engineered code. Among other findings we discovered that, on average, identifiers are around 10% *longer* and comments 90% *shorter* in engineered vs. non-engineered code; method names with the word *test* — evidence of the use of automated tests — are much more prevalent in engineered code; and comments that warn developers about matters or code to be handled with care — so-called *notice* comments — are twice as common in engineered code. To the best of our knowledge this is the study that analyzes the largest number of identifiers and comments to date.

1. INTRODUCTION

Identifiers and comments are important sources of information to understand software. Approximately 70% of the code of a system consists of identifiers [1]. Identifier length and quality, as well as naming conventions overall, influence the difficulty and time required to complete maintenance tasks like code reading or debugging [2–4]. After the code itself, comments are the main source of documentation for software maintainers [5], and experiments show that commented code is easier to understand when compared to uncommented code [6]. Therefore these elements are of paramount importance for software development in general.

In this paper, we focus on answering whether comments and identifiers are related to the use of well-established software engineering (SE) practices. Previous work has demonstrated that naming conventions and commenting practices vary by project, over time, and even between proprietary and open-source software [7]. We broaden this inquiries to ask: Do developers who adopt other good engineering practices tend to name things in code differently than developers who do not? Similarly, are comments in code where these practices were applied different from comments in code where they were not? In particular, what types of comments are more common in engineered vs. non-engineered code? We present a large-scale empirical study to gather evidence about these questions. Compared to related work our investigation is *broad* in the sense that we target *essential and general characteristics* of identifiers — such as length and format — and comments — such as length and type — in engineered vs. non-engineered code. In particular, we focus on software developed within open source code repositories, in particular on GitHub, and using Java, the second most active language on that platform[1].

We collect data by applying a framework proposed by Munaiah et al. [8] that curates engineered software projects from GitHub. The approach helps identify repositories that contain evidence of the use of solid SE practices, proposing means for validating their existence. The researchers define the following seven dimensions: (1) *community*, as evidence of collaboration; (2) *continuous integration*, as evidence of quality; (3) *documentation*, as evidence of maintainability; (4) *history*, as evidence of sustained evolution; (5) *issues*, as evidence of project management; (6) *license*, as evidence of accountability; and (7) *unit testing*, as evidence of quality. By using these dimensions as features, they apply machine learning algorithms trained with known engineered projects to produce a classifier able to predict whether or not a given repository contains an engineered software project.

Based on this model, we define *engineered code* as code contained in projects classified as *engineered* by Munaiah et.'s approach [8]. Conversely, *non-engineered code* is code

[1] https://github.info/ - 6/12/2019.

SAC'20 Brno, Czech Republic March 30-April 3, 2020

DOI: 10.1145/1235

contained in projects classified as non-engineered by the same approach. We gathered 550 projects classified as engineered, and approximately 10k classified as non-engineered. This difference in number of projects is due to the fact that engineered projects tend to be much larger — *e.g.*, in terms of number of methods — than non-engineered projects. The dataset thus amounts to approximately 1M methods in each category. We parsed all code and extracted around 3M identifiers and 3M comments from each repository, forming a dataset of 6M identifiers and 6M comments.

From identifiers we collected several features, such as length, format (*e.g.*, camel case and snake case), number of words, and whether the identifier contains only English words. We analyzed identifiers separately – class, interface, field, method, and variable names – since each has its own particularities. For comments we looked at common features and also adopted a machine learning approach that classifies each in one of six categories established by Pascarella and Bacchelli [9]: (1) *purpose*: comments used to describe the functionality of linked source code; (2) *notice*: comments related to the description of warning, alerts, messages, or in general, functionalities that should be used with care; (3) *under development*: comments that refer to temporary tips, notes, or suggestions that developers use during development (*e.g.*, `TODO`); (4) *style & IDE*: comments that are used to logically separate the code or provide special services (may be added automatically by the IDE or used to communicate with it); (5) *metadata*: comments that define meta information about the (e.g., authorship comments with the tag `@author`); and (6) discarded: comments that do not fit into any of the previously defined categories. We then analyzed the distribution of types of comments among engineered and non-engineered projects.

Findings. Our study provides evidence for the following:

1. identifiers are significantly different in engineered and non-engineered code. In particular, we found that these are around 10% longer in engineered code and that identifiers with formats non-conformant with common Java conventions — such as class names starting with lower case letters — are much more frequent in non-engineered code. We also found that several concepts are common in identifiers of one group and not the other. For instance, methods containing the word *test* are very common in engineered but not in non-engineered code; and

2. comments are also significantly different in engineered and non-engineered code. In particular, we found that these are around 90% shorter in engineered code and that more sophisticated comments — such as the ones that warn developers about functionalities that should be used with care — are twice as prevalent in engineered code.

We have also found some counterintuitive results. For instance, classes with sample names such as `Foo` and `C` were very common in engineered code. Moreover, although comments generated by IDEs (with the tag *Auto-generated*) are left more frequently in non-engineered projects, we found more than 4,000 such comments in engineered code. Our study also establishes a large dataset of 6M identifiers and 6M comments in engineered and non-engineered code that in the future can be used by researchers in several types of experiments (*e.g.*, it can be used for training different types of classifiers).

Outline. The remainder of this paper is structured as follows. Section 2 provides background information about the topics targeted in this paper; Section 3 outlines the experimental setting of our study; and Section 4 presents the results of our experiment and its in-depth analysis. Section 5 discusses limitations of our study; and Section 6 summarizes published research related to ours. Finally, Section 7 presents the main conclusions of our experiment.

2. BACKGROUND

In this paper we analyze identifiers and comments in engineered and non-engineered code. We adopt a framework proposed by Munaiah et al. [8] to curate our samples of code likely developed in accordance with high-quality SE practices, and code that likely was not developed following such practices.

Munaiah et al. [8] proposed a framework and presented a reference implementation as a tool called `reaper`, to enable researchers to select GitHub repositories that contain evidence of an *engineered* software project. By using the seven aforementioned dimensions — *community*, *continuous integration*, *documentation*, *history*, *issues*, *license*, and *unit testing* — and manually classified data sets of repositories, they trained classifiers capable of predicting whether a given GitHub repository contains an engineered software project.

To evaluate each of the seven dimensions, the authors define specific metrics. For *community* they use *core contributors*, which is the cardinality of the smallest set of contributors whose total number of commits to a source code repository accounts for 80% or more of the total contributions. For *continuous integration* they use a boolean *CI* metric that tells whether the repository applies continuous integration by looking for a configuration file in the source code repository (the metric currently supports Travis CI, Hound, Appveyor, Shippable, MagnumCI, Solano, CircleCI, and Wercker). For *documentation* the authors propose *comment ratio*, that is, the ratio of the number of comment lines of code to the number of non-blank lines of source code in a repository. For *history* they use *commit frequency*, which is the average number of commits per month. For *issues* the authors propose *issue frequency*, which is the average number of issue events transpired per month. For *license* the authors use a boolean metric to tell whether or not the repository contains a popular license. This is done by using the GitHub License API, which identifies the presence of popular open source licenses by analyzing files such as LICENSE and COPYING in the root of the source code repository. Finally, for *unit testing* the authors look at *test ratio*, which is the ratio of number of source lines of code in test files to the number of source lines of code in all source files.

The authors used two types of classifiers: Score-based, a custom approach to implement the evaluation framework that allows complete control over the classification; and Random Forest, a tree-based approach to classification in which multiple trees are trained such that each tree casts a vote which is then aggregated to produce the final classification. They also used two different datasets for training: *organization* — a set of 300 repositories with engineered code owned by organizations such as Amazon, Apache, Facebook, Google, and Microsoft; and *utility* — a set of 300 repositories with engineered code implementing general-purpose utility to users other than the developers themselves. Such an approach was used in our study to separate samples of engineered and non-engineered code.

To investigate comments, we begin from the observation that not all comments available in software have the same goal and intended target audience. To verify what types of comments are more or less prevalent in engineered versus non-engineered code, it is useful to be able to classify them. Pascarella and Bacchelli [9] produced a taxonomy of source code comments and subsequently investigate how often each category occur by manually classifying more than 2,000 code comments from six diverse open source Java projects. Such manual classification was then used to train a classifier able to predict the type of a given comment.

Pascarella and Bacchelli's [9] taxonomy consists of the following six top categories referred before: (1) *purpose*: comments used to describe the functionality of linked source code; (2) *notice*: comments related to functionalities that should be used with care (*e.g.*, comments to warn the developers of a deprecated functionality); (3) *under development*: comments related to ongoing and future development tasks (*e.g.*, comments containing the `TODO` tag and commented code); (4) *style & IDE*: comments that are used to logically separate the code or provide special services (*e.g.*, comments added by the IDE or used to communicate with it); (5) *metadata*: comments that define meta information about the (*e.g.*, authorship comments with the tag `@author`); (6) *discarded*: comments that do not fit into any of the previously defined categories. They also refine each category into subcategories, but in this paper we focus only on the top-level classification described above.

3. STUDY SETUP

Our goal is to investigate the impact of software engineering practices on how developers name elements and add code comments. Therefore, for our study, we are interested in the following research questions. In the context of large-scale Java source code repositories:

RQ1. Are *identifiers* significantly different in engineered versus non-engineered code? If so, what are these differences?

RQ2. Are *comments* significantly different in engineered versus non-engineered code? If so, what are these differences?

For features that admit direct numerical comparisons — *e.g.*, length of the identifiers — we have as many hypotheses as features used in our experiment, since we want to check each one separately. Therefore, we define our experiment *null* and alternative hypotheses for these features as follows. For each identifier or comment feature F_i:

- H_i-0: there is no significant difference in F_ii between identifiers/comments in engineered and non-engineered code;
- H_i-A: there is a significant difference in F_i between identifiers/comments in engineered and non-engineered code.

We compare remaining features descriptively and qualitatively, described inline.

3.1 Datasets

Our study requires a large quantity of code from projects classified as engineered and non-engineered, respectively. We aimed to generate samples that each contained around 1M methods, to produce a balanced dataset. Our procedure consisted in taking Java projects classified as engineered and non-engineered from the application of Munaiah et al.'s [8] work. Fortunately, the authors of that work made available a large `csv` file generated by `reaper`, their curation tool, classifying 1,857,423 GitHub repositories[2]. We used this data as our oracle for determining whether a project was engineered or not when drawing samples.

From the `csv` file, we selected all Java projects classified as engineered and non-engineered for both training datasets (utility and organization). By looking at the average number of methods per engineered project, we sampled 550 that appeared in the `csv`, to arrive at the approximate number of 1M. We did the same for non-engineered projects, sampling the ones classified as non-engineered. Since these contain fewer methods per project on average, we collected 10K to reach approximately 1M methods. Overall, this produced a sample of projects with 1,165,025 engineered methods and 1,028,109 non-engineered methods. Since the difference was not too large — around 12% — we decided to keep all the data and used both samples as they were.

3.2 Identifier Analysis

To analyze identifiers we first extracted all of them from each repository. To do that we needed a Java parser capable of pinpointing identifiers in a given Java file. In particular we adopted javaparser, "a simple and lightweight set of tools to generate, analyze, and process Java code"[3]. As commented before we also wanted to analyze each type of identifier — class, interface, field, method, and variable names — separately, as each one has its own particularities. In this way we can have a more precise analysis and investigate specific features for each type. For instance, since in Java it is common to name interfaces starting with a capital *I*, we extracted such a feature for interface names; since it is also considered a good practice to include at least one *verb* in method names, we extracted such a feature for method names. Table 1 shows the total number of identifiers collected for our experiment. Note that there are approximately 3M identifiers for each class.

Table 1: Total number of identifiers collected for our experiment by type and class (eng = engineered; neng = non-engineered).

type	eng	neng
class	145,402	176,650
interface	14,617	10,402
field	460,673	962,981
method	1,069,602	925,452
variable	1,040,834	976,359
total	2,731,128	3,051,844

One of the features common to all identifiers that we wanted to investigate was its *case format*, since there are common patterns followed to name things in code with respect to case, in particular for Java code. The most popular case format is probably *camel case*, which consists in writing compound words or phrases such that each word in the middle of it begins with a capital letter, with no intermediate spaces or punctuation (*e.g.*, `CamelCase` or `camelCase`). Another common format is *snake case*, which separates elements of compound words or phrases with an intermediate underscore character (*e.g.*, `Snake_Case` or `snake_case`). Since

[2] https://reporeapers.github.io/ - 4/1/2019.

[3] https://javaparser.org/ - 6/5/2019.

both camel and snake case might start with lower or upper case letters, we also made this distinction in our categorization. Some identifiers can also start with a leading underscore or contain only lower case letters.

To cover all these alternatives and others with respect to case formatting, we thus classified each identifier using the following eight format categories: (1) *lead underscore*: starts with an underscore character; (2) *upper underscore*: snake case with upper case letters; (3) *lower underscore*: snake case with lower case letters; (4) *lower camel*: camel case starting with a lower case letter; (5) *upper camel*: camel case starting with an upper case letter; (6) *all lower*: all lower case letters (except for other valid characters such as numbers); (7) *all upper*: all upper case letters (except for other valid characters such as numbers); and (8) *unknown*: does not match any of the above. The algorithm we used to classify identifiers into one of the defined categories is defined in Algorithm 1[4]. To detect camel casing we used the following regular expressions: `[a-z]+[A-Z]+w+` (lower camel), `[A-Z]+[a-z]+w+` (upper camel).

Algorithm 1 Function to detect case format

```
1: function caseFormat(String s)
2:     if s contains underscore then
3:         if s starts with an underscore then
4:             return LEAD_UNDERSCORE;
5:         if s's letters are all lower case then
6:             return LOWER_UNDERSCORE;
7:         if s's letters are all upper case then
8:             return UPPER_UNDERSCORE;
9:     if s's first character is lower case ∧ s is camel case then
10:        return LOWER_CAMEL;
11:    if s's first character is upper case ∧ s is camel case then
12:        return UPPER_CAMEL;
13:    if s's letters are all lower case then
14:        return ALL_LOWER;
15:    if s's letters are all upper case then
16:        return ALL_UPPER;
17:    return UNKNOWN;
```

Identifier features. The common features we extracted for each identifier were the following:

1. Length: the size of an identifier in number of characters;

2. Format: the case format as categorized before;

3. Number of words: the number of words by splitting identifiers according to their format (*e.g.*, CamelCase is formed by two words: *camel* and *case*; snake_case is formed by two words: *snake* and *case*);

4. English: a boolean feature that asserts whether the identifier contains only English words. For this feature we used a comprehensive English dictionary also containing technical computing — termed *hacker* by SCOWL (Spell Checker Oriented Word Lists) — terms[5] (*e.g.*, *grepped*). To be able to extract this information we also split identifiers according to their case format and check each word.

[4]Based on a solution given in the Google Guava community (https://github.com/google/guava/issues/2212#issuecomment-358368610 – 06/15/2019).

[5]http://wordlist.aspell.net/ - 6/5/2019.

For method names we also collected a *contains verb* boolean feature, which verifies whether the name contains at least one word that can be considered a verb in English. For this feature we used WordNet[6] and adopted a conservative approach: when the identifier contains at least one word that can be classified as a verb in WordNet, we consider it to contain a verb. For interface names we collected three additional features: (1) whether the identifier starts with a capital *I*; (2) whether it ends with *ible*; and (3) whether it ends with *able* (common patterns used to name interfaces). Finally, for field and variable names, we included two additional features: (1) whether it contains an underscore; and (2) whether it is formed by a single character (these features can be computed by looking at other features but we decided to look at them separately to make the analysis easier). Additionally, we measured the *most common identifiers* of each type and the *most common words* that appear in these identifiers.

3.3 Comment Analysis

Not all the comments have the same goal and target audience [9]. Therefore, to investigate comments in our study, and how SE practices impact them, we focused on categorizing them and looking at their distribution in engineered and non-engineered code. To classify comments according to their goal and target audience, we adopted the machine learning classifier proposed by Pascarella and Bacchelli [9] mentioned earlier. We first extracted all comments from the source code of the group of engineered and non-engineered projects by using javaparser, the same parser mentioned before. Table 2 shows the total number of comments collected for our experiment. Again there are around 3M comments for each class.

Table 2: Total number of comments collected for our experiment by class (eng = engineered; neng = non-engineered).

	eng	neng
total comments	2,688,834	3,289,900

We implemented a Random Forest classifier in Python with the scikit-learn package [7] using the same features established by Pascarella and Bachelli. Some of these features are presented in Table 3. Note that there are several characteristics extracted from comments to help the classifier to determine the correct category. We developed a Java application to extract these features for the comments collected in our study. Then, we trained the classifier with the same training data used by the authors and ran the classifier with our comments' feature values to obtain the classification for each.

We also built a relational database with all extracted comments and their given classification. As will be commented below, this database will be available for other researchers together with all other important experimental artifacts generated in our study. For comments we also looked at *length* — the size of the comment content in number of characters —, and other common characteristics.

3.4 Statistical Analysis

For the features where direct comparisons can be made, we ran statistical tests to verify whether the obtained differ-

[6]https://wordnet.princeton.edu/ - 6/5/2019.

[7]https://scikit-learn.org/ - 6/17/2019.

Table 3: Some of the features extracted from comments to feed into the machine learning classifier adopted in our study [9].

Feature	Type	Description
words	numeric	counts the occurrence of each word in the bag of unique words
punctuation	boolean	used in combination of a regular expression to distinguish source code from natural language
words count	numeric	measures the length of the comment, using the words as unit size
unique words count	numeric	measures the length of the comment, only unique words are counted
deprecation	boolean	true if a comment contains special tags like `@deprecation`
usage	boolean	true if a comment contains special tags such as `@usage`, `@return` or `@value`
exception	boolean	true if a comment contains special tags such as `@exception` or `@throws`
TODO	boolean	true if a comment contains keywords such as todo or fix or a link to a bug is detected
incomplete	boolean	true if a comment contains an empty body
commented code	boolean	true if a comment contains code snippets
directive	boolean	true if a comment contains special sequence of symbols used by IDE
formatter	boolean	true if a comment is composed of patterns of symbols or characters
xlicense	boolean	true if a comment contains words such as license, copyright, legal or law
ownership	boolean	true if a comment contains tags such as `@author` or `@owner`
pointer	boolean	true if a comment contains a reference to an external linkable resource
automatic generated	boolean	true if a comment contains text automatically inserted by IDE e.g., Auto-generated method stub

ences where significant. Since our data in general does not seem to follow a normal distribution, as analyzed by looking at Q-Q plots, we decided to run Wilcoxon signed rank tests. For the other features we compared outcomes descriptively.

3.5 Dataset Accessibility and Reproducibility

We welcome other researchers to reproduce and replicate our study on the same or different settings. We will make available all the necessary artifacts, including raw data, scripts, and source code used to run the experiment. All artifacts will be made available at the following URL: *http://github.com/ommited.for.double.blind.review.*

4. RESULTS AND ANALYSIS

In this section we present the results for our two research questions: RQ1. are *identifiers* significantly different in engineered versus non-engineered code? If so, what are these differences?; and RQ2. are *comments* significantly different in engineered versus non-engineered code? If so, what are these differences? We start by looking at identifiers.

Identifier analysis. Table 4 shows our main results for all identifiers. For *length* — number of characters of the identifier — and *number of words*, we looked at mean values; and for *English* — a boolean feature that tells whether the identifier is formed only by English words — we look at percentages of identifiers for which the value of the feature is true. Note that for most types of identifiers, these are *longer* in engineered code — around 10% longer on average —, except for field names which are around 31% longer in non-engineered code. The same happens for number of words: on average, identifiers in engineered code have around 10% more words when compared with non-engineered code; except for field names which contain 10% more words in non-engineered code. This outcome is interesting: it shows that, in general, developers tend to use names for identifiers that are longer and possess more words in engineered projects when compared to non-engineered projects (an exception are field names: it appears that developers not using solid SE practices tend to be *wordy* when naming these). A general result for both classes of code is that variable names tend to be shorter than all other types of identifiers. The lengths of the latter range from around 11–17 characters while the lengths of variable names range from around 6-7 characters.

Note that most identifiers collected in our study contain only English words, which can be considered a good practice (except when developers are using a different language for their projects). We found that engineered code has, on average, around 77% of identifiers consisting only of English words, 8% more when compared to non-engineered code. Note that around 30% of all identifiers in non-engineered code are not formed only by English words. To check the statistical significance of the differences we observed for the *length* feature we ran statistical tests for each type of identifier. All tests indicated a significance difference at the 0.01 level, which indicates that these differences are indeed significant.

Table 5 shows our analysis of case format frequency of identifiers. The table shows the frequency of each type of case format for each type of identifier. We highlight in **bold** the most frequent formats and in *italics* the most discrepant outcomes when comparing engineered and non-engineered code. Note that for all types of identifiers, the most common case formats coincide in both engineered and non-engineered code, and these follow well-established Java format conventions. For instance, note that for class names the most common format is *upper camel*; and for methods, *lower camel*. However, we can see that non-engineered code tend to deviate from these conventions much more often. For instance, engineered class names are formatted with *upper camel* 96.63% of times, so only 3.37% of identifiers deviate from this common convention. In non-engineered code, this happens much more often: 17.86% of the times. We also found large discrepancies between some of the identifier types. Non-engineered code contain 11.34% of class names following the *all lower* format, which can be considered a bad practice since class names generally should start with an upper case letter. Field names following the *lower underscore* format are also frequent in non-engineered code: they appear 16.65% of the times (while only 2.09% for engineered code).

For method names we analyzed the identifiers that contained only English words and did not possess verbs with the *contains verb* feature. This can be considered a bad practice as it is usually recommended that method names contain at least one verb term to describe its performed action or behavior. We found that 9.73% of method names in engineered code did not possess any term that could be considered a verb against 10.30% in non-engineered code. This indicates that methods without verb terms tend to occur more often in non-engineered code (around 6% more frequently). With respect to interface naming, we found that 12.77% of the

Table 4: Main results for our identifier study (eng = engineered; neng = non-engineered; mean # words = mean number of words; %english = percentage of identifiers that contain only English words; diff%: difference in percentage).

	mean length			mean # words			%english		
type	eng	neng	diff%	eng	neng	diff%	eng	neng	diff%
class	17.55	11.73	33.17	2.97	2.03	31.72	71.28	69.82	2.05
interface	15.71	13.36	14.95	2.65	2.30	13.27	75.36	74.83	0.71
field	11.91	15.70	-31.84	1.98	2.19	-10.54	74.72	55.40	25.86
method	13.67	11.60	15.20	2.57	2.32	9.61	85.49	80.80	5.49
variable	7.15	5.85	18.20	1.46	1.34	8.23	80.12	73.80	7.88
mean	13.20	11.65	9.93	2.33	2.04	10.46	77.40	70.93	8.40

Table 5: Case format frequency in percentage for each type of identifier (us = underscore; eng = engineered; neng = non-engineered).

	lead us		upper us		lower us		lower camel		upper camel		all lower		all upper		unknown	
type	eng	neng	eng	neng	eng	neng	eng	neng	eng	neng	eng	neng	eng	neng	eng	neng
class	0.04	0.12	0.03	0.12	0.02	0.63	0.10	1.19	**96.63**	**82.14**	*0.34*	*11.34*	1.20	2.26	1.63	2.20
interface	0.02	0.00	0.01	0.15	0.00	0.83	0.05	0.93	**98.42**	**94.93**	0.15	1.27	0.50	0.91	0.85	0.97
field	1.97	1.05	15.54	8.18	*2.09*	*16.82*	**41.28**	**30.06**	0.52	2.64	**29.81**	**24.33**	5.43	3.06	3.38	13.85
method	0.11	0.10	0.02	0.02	0.75	1.22	**78.98**	**76.17**	0.29	1.74	**16.46**	**17.81**	0.02	0.16	3.38	2.78
variable	0.32	0.51	0.18	0.27	0.41	2.34	**33.84**	**23.75**	0.17	0.68	**63.86**	**70.37**	0.19	0.74	1.04	1.34

identifiers in engineered code and 13.46% of the identifiers in non-engineered code started with a capital *I*. This indicates that such a pattern is followed around 5% more often in non-engineered code. For the other patterns, we found that only 5 interfaces ended with -ible in each category; and 323 in engineered and 290 in non-engineered ended in -able. Therefore, for interfaces we did not find much of a difference between engineered and non-engineered code with respect to common naming patterns.

We also analyzed the most common identifier names and words that appear in these identifiers. Here we focus on class and method names, and words that appear in these identifiers. Table 6 shows the top 20 most common class and method names from engineered and non-engineered code. Note that some class names are common in both classes. `Main`, `User`, and `Node` are examples of class names that are frequent in both engineered and non-engineered code. However, some identifiers are common in one of the groups only. For instance, `R` — a common class in Android applications —, `style`, and `color` are common in non-engineered but not in engineered code (note the examples of classes named with the *all lower* format, which as commented before could be considered a bad practice). On the other hand it is intriguing to see class names used in sample code — such as `Foo`, `A`, `B`, and `C` — very frequently in code where solid SE practices where applied.

With respect to method names we observe a similar pattern: some commonalities between engineered and non-engineered code but interesting deviations. Common `Object` method implementations such as `toString`, `equals`, and `hashcode` are common in both groups. However, there are some important differences. For instance, the most common method name in non-engineered code is the `main` method, which does not appear among the top 20 most common method names for engineered code. It is interesting to observe that among common identifiers in non-engineered code are Android app elements, such as the `R` class commented before, and the `onClick` method.

Table 7 shows the top 20 most common words that appear in class and method names from engineered and non-engineered code. Again we see some commonalities but important differences. `impl`, `listener`, and `data` are terms frequently used in class names for both groups. `test` the most common word in engineered class names is also frequent in non-engineered code, but 5x less. Terms that indicate more sophisticated behavior such as `factory`, `file`, and `map` are common in engineered code, but not in their non-engineered counterpart. For method words it is interesting to note that the term `test` — which indicates the use of automated tests — appear frequently in engineered code, but not in non-engineered code.

With respect to research question 1 our results indicate that identifiers are significantly different in engineered and non-engineered code. In particular, we found that these are around 10% longer in engineered code and that identifiers with formats non-conformant with common Java conventions — such as class names starting with lower case letters — are much more frequent in non-engineered code. Thus we observe that SE practices impact significantly on the way developers name elements in code.

Comment analysis. We now turn into our comments investigation. Before we look into the distribution of the different types of comments found in our study, we compare the *lengths* of comments in engineered and non-engineered code. We found that the mean length of comments in number of characters was 94.92 for engineered code and 181.95 for non-engineered code. This indicates that comments in non-engineered code tend to be much longer — around 92% — when compared to comments in engineered code. This is somehow surprising as one could expect comments to be more comprehensive in engineered code. However, we must also take into account that comments may sometimes be superfluous or verbose. The idea is that when code is sufficiently well-written it tends to be self-explanatory and thus eliminate the need for comments (or, at least make them leaner). There is a famous quote by Martin Fowler that asserts that "when you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous" [10]; and also "good code is its own best docu-

Table 6: Top 20 most common class and method names in engineered and non-engineered code (eng = engineered; neng = non-engineered; freq. = frequency).

	class names				method names			
	eng		neng		eng		neng	
#	name	freq.	name	freq.	name	freq.	name	freq.
1	Builder	1144	R	2126	toString	13172	main	22083
2	A	529	string	2007	run	6816	actionPerformed	14117
3	B	182	attr	200	equals	6027	onClick	11962
4	Messages	157	drawable	1986	hashcode	5505	onCreate	11471
5	ConcreteBuilder	169	MainActivity	1966	visit	5207	run	11063
6	ObjectFactory	140	id	1917	getName	4749	toString	10823
7	Foo	126	layout	1884	get	4248	getName	4105
8	User	119	BuildConfig	1559	setUp	4012	onCreateOptionsMenu	3928
9	Util	116	style	1514	create	3845	equals	3765
10	C	111	Main	1308	actionPerformed	3590	getId	3747
11	Main	100	dimen	1221	apply	3137	init	3130
12	TestModule	99	menu	1123	init	3019	update	3027
13	Activator	96	color	729	append	2979	setId	2895
14	Externalizer	95	ApplicationTest	629	getId	2970	hashCode	2717
15	Person	93	styleable	599	execute	2814	onOptionsItemSelected	2627
16	SavedState	92	Solution	599	accept	2720	setName	2328
17	Node	90	integer	488	add	2690	mouseClicked	2121
18	MainActivity	83	anim	433	remove	2561	add	2051
19	Utils	78	User	429	getValue	2529	initComponents	2037
20	Request	77	Node	425	appendFields	2500	visit	1995

Table 7: Top 20 most common words appearing in class and method names in engineered and non-engineered code (eng = engineered; neng = non-engineered; freq. = frequency).

	class words				method words			
	eng		neng		eng		neng	
#	name	freq.	name	freq.	name	freq.	name	freq.
1	test	23203	activity	6881	get	241415	get	215821
2	type	4459	main	4420	set	118464	set	111971
3	handler	3236	test	4084	test	77566	on	75696
4	impl	3122	list	3039	is	46430	create	36566
5	exception	3017	string	2662	to	38691	to	26363
6	list	2994	listener	2643	create	38073	action	25263
7	map	2888	view	2418	on	29282	is	22950
8	service	2855	my	2390	name	25514	id	22924
9	builder	2795	impl	2376	string	22873	main	22639
10	factory	2744	id	2277	type	22532	performed	21448
11	abstract	2621	r	2126	with	22021	name	17931
12	data	2530	config	2110	add	21934	string	17234
13	action	2046	service	2100	value	17985	add	16241
14	property	2036	layout	2078	id	17341	click	15894
15	listener	2007	drawable	2054	from	14196	item	13204
16	set	2000	menu	2044	for	12125	run	11999
17	command	1832	data	2040	remove	11878	update	11511
18	check	1819	attr	2022	visit	11577	init	10480
19	token	1815	adapter	2005	update	11020	list	10086
20	file	1800	user	1962	list	10918	value	9839

mentation" [11]. The fact that non-engineered code contains longer comments on average aligns with this idea. We ran a Wilcoxon test to check whether the length difference was statistically significant. The test indicated a significant difference at 0.01 level.

Table 8 presents the distribution of the different types of comments in percentage for engineered and non-engineered code. Note that the most prevalent type of comment is *purpose* — comments used to describe the functionality of linked source code — in both groups. However these are more prevalent in non-engineered code (4.61% more frequent). This seems to support the idea that non-engineered code contains more explanatory comments than engineered code. The frequency of comments related to ongoing and future development tasks — *under development* — is very similar for both groups. This is somehow surprising as one

would expect comments such as these to be more common in non-engineered code as they include things such as commented code.

For the remaining three types of comments the difference is much larger: all of them are around twice as frequent in engineered code. This indicates that developers using SE practices tend to make use of more sophisticated types of comments much more often: comments related to the description of warning, alerts, messages, or, in general, functionalities that should be used with care — *notice*; comments used to logically separate the code or provide special services — style and IDE; and comments that define meta information about the code, such as authors, license, and external references — *metadata*. For instance, *authorship* comments containing the tag `@author` are around 38% more frequent in engineered code. And comments containing the term *license* are around 8x more frequent in engineered code. Another interesting type of comment is the one added automatically by tools with the special term *Auto-generated*'. These types of comments were 8x more frequent in non-engineered code.

Table 8: Frequency of each type of comment in percentage for both groups (eng = engineered; neng = non-engineered; diff = difference in percentage).

	eng	neng	diff
purpose	85.89	89.85	-4.61
under development	5.81	5.75	0.89
notice	4.58	2.31	49.45
style and IDE	2.41	1.37	42.97
metadata	1.28	0.66	48.36

To have a better idea about the differences in the distribution of the types of comments across groups, Figure 1 shows a barplot for the three types of comments where we encountered the largest differences.

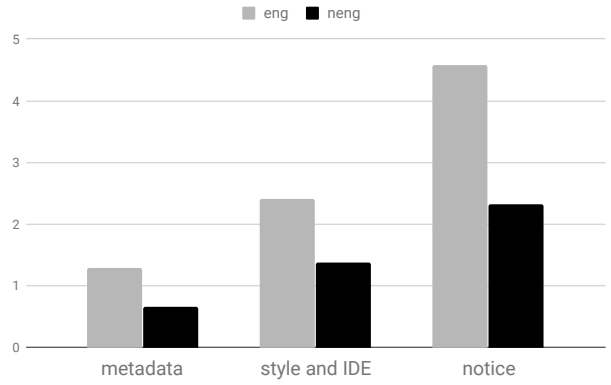

Figure 1: Grouped bar plot of the frequency of some types of comments in percentage for both groups (eng = engineered; neng = non-engineered).

With respect to research question 2 our results indicate that comments are also significantly different in engineered and non-engineered code. In particular, we found that these are around 90% shorter in engineered code and that more sophisticated comments — such as the ones that warn developers about functionalities that should be used with care — are twice as prevalent in engineered code. Thus we observe that SE practices impact significantly on the way developers comment code.

5. STUDY LIMITATIONS

Our study presents limitations that must be taken into account. Such limitations might affect its generalization. In particular our analyses are based on Java and thus does not generalize to other types of language and technology. We believe — and, in fact, encourage — similar studies involving other languages such as Python and C would help establish more general conclusions about differences in identifiers and comment between engineered and non-engineered code.

Pascarella and Bacchelli's [9] report that their approach to classify comments adopted in our study yields a precision of around 80%. This means that our comments analysis can contain a 20% of misclassified data. In the future we can apply improved versions of that approach — *e.g.*, training it with larger sets of data — to be able to circumvent this imprecision.

6. RELATED WORK

Munaiah et al. [8] propose a model that identifies well-engineered code, to distinguish it from various other types of code found in open-coding hosting sites. Their particular aim is to help researchers curate repositories from open coding forges, in particular GitHub[8]. This challenge of identifying useful repositories for mining is well-motivated [12]: many repositories on GitHub do not contain useful code, or even code at all [13, 14]. We use Munaiah et al.'s model and the associated dataset to conduct an empirical study on the differences between naming and commenting practices and choices made in such engineered projects versus those that are not. Unlike the prior work, we do not propose a model that distinguishes between such datasets *per se*. It is possible that differences in variable uses may provide another useful signal to such models. We leave such an investigation to future work, as well as corroborating studies that could be performed using other curated repositories [15, 16] or frameworks to help construct them (like GHTorrent, Boa [17], or Orion [18].

Identifiers and comments are well-studied as important sources of information for developers trying to read, write, or maintain source code. Both comment quality and accuracy, especially as programs evolve, are important to program comprehension [19, 20]. Researchers have conducted a number of lab studies, with professional and student developers, to characterize identifier relationship to program comprehension and other maintenance tasks like modification and debugging. Identifier names in general are key to program comprehension [21–23]. Our results corroborate or are otherwise consistent with some of the phenomena previously studied in lab settings. For example, we find in general that engineered projects contain, on average, longer identifier names and names that more closely correspond to En-

[8]https://github.com/ - 06/17/2019.

glish prose. In experimental settings, full word identifiers, and longer identifiers generally, appear to result in better and faster comprehension than shorter identifiers [2, 24]. Longer, more descriptive names can also support more effective debugging [4].

The effect of variable names appears to vary by developer experience, though the direction of the effect is not consistent between studies. However, our finding that engineered projects are slightly more likely to follow a camel-casing naming convention is consistent with work that shows that this naming convention is more accessible for beginners [3]. This suggests that well-engineered projects may have an easier time onboarding new developers. Indeed, naming conventions are important for program comprehension generally and developer onboarding in particular [25], but there is wide variance in the degree to which developers and project adhere to specific conventions [7, 26]. We provide large-scale empirical evidence that conformance to particular conventions may correlate with the following of other good engineering practices in a project.

We evaluate the connection between projects developed following good engineering principles and various characteristics about identifier selection and commenting practices. This, at least indirectly, speaks to a potential relationship between naming practices and code. Others have looked at this question directly, finding a relationship between poorly-named code and static analysis warnings [27]. Comment quality may also correlate with code quality [28]. Inconsistent naming may offer refactoring opportunities [26] or pointers to possible defects [29]. Our work is descriptive rather than normative, but we identify relationships between naming schemes and engineering quality overall that may be suggestive for engineering practice generally. Such engineering efforts may benefit from orthogonal work on suggesting good, informative variable names, using formal models [1] or, more recently, machine learning [30, 31].

All such studies, including those like our own, additionally benefit from more accurate NLP mechanisms to split identifiers into component words [32]. In fact, in our study we adopted a simple approach to split identifiers using the well-known case formats camel case and snake case. To detect verbs we also adopted a conservative and simple approach using WordNet. However, there are more sophisticated approaches that perform part-of-speech tagging on identifiers (such as the one presented by Gupta et al. [33]). In the future we intend to incorporate more accurate analyses to attain more precise results with respect to identifier wording and part-of-speech tagging.

7. CONCLUSIONS AND FUTURE WORK

In this paper we present a large-scale study comparing identifiers and comments in code where SE practices were applied and code where they were not. We discovered evidence that both identifiers and comments are significantly different in engineered versus non-engineered code. In particular, we found that identifiers are around 10% longer in engineered code and that identifiers with formats non-conformant with common Java conventions — such as class names starting with lower case letters — are much more frequent in non-engineered code. We also found that comments are around 90% shorter in engineered code and that more sophisticated comments — such as the ones that warn developers about functionalities that should be used with care — are twice as prevalent in engineered code. Thus we observe that SE practices impact significantly on the way developers name structures and comment code.

One might argue that our study only confirms the common wisdom that applying SE conventions result in better naming and commenting practices. Even if that was the case it would still be useful as we show these intuitions empirically hold in a Java open source setting. On the other hand, we believe we have found interesting counterintuitive outcomes. For instance it is quite surprising that comments are, on average, 90% shorter in engineered code as one would expect engineered code to possess more comprehensive comments. Also, we found sample class names such as `Foo` much more often in code where SE practices were applied (in fact, these appear among the top 20 most common class names in that group). Another interesting observation is that although identifiers non-conformant with common Java conventions are more common in non-engineered code, this also happens quite frequently in engineered code. For example, there were more than 600 classes with names starting in lower case letters in this group.

In the future we intend to further analyze our results to look for additional finer-grained differences that can be found in identifiers and comments in engineered and non-engineered code. For instance, we plan to apply Pascarella and Bacchelli [9]'s approach to classify code into the finer-grained categories established by those authors. In fact, for each top category adopted in our study, there are subcategories that can be detected for each comment (for instance, within the top category *purpose*, there are two subcategories: (1) summary — a comment that explains *what* some code does; and (2) expand — a comment that explains *how* some code works). This would help better understanding differences in comments between engineered and non-engineered code.

Acknowledgements

Otavio Lemos would like to thank FAPESP for financial support (grant 2017/27098-1) and Rodrigo Barros for machine learning consulting. We also thank Vaibhav Saini who walked us through the use of their metric extraction tool used in this investigation.

References

[1] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, Sep. 2006.

[2] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ser. ICPC '06, 2006, pp. 3–12.

[3] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empirical Softw. Engg.*, vol. 18, no. 2, pp. 219–276, Apr. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-012-9201-4

[4] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 31–40.

[5] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting*

&Amp; Designing for Pervasive Information, ser. SIGDOC '05. New York, NY, USA: ACM, 2005, pp. 68–75.

[6] T. Pawelka and E. Juergens, "Is this code written in english? a study of the natural language of comments and identifiers in practice," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 401–410.

[7] D. Lawrie, H. Feild, and D. Binkley, "Quantifying identifier quality: An analysis of trends," *Empirical Softw. Engg.*, vol. 12, no. 4, pp. 359–388, Aug. 2007.

[8] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Softw. Engg.*, vol. 22, no. 6, pp. 3219–3253, Dec. 2017.

[9] L. Pascarella and A. Bacchelli, "Classifying code comments in Java open-source software systems," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 227–237.

[10] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] S. C. McConnell, *Code Complete: Book and Online Course Bundle*, 2nd ed. United Kingdom: C.B.Learning, 2010.

[12] N. Nagappan, "Potential of open source systems as project repositories for empirical studies (working group results)," in *Proceedings of the 2006 International Conference on Empirical Software Engineering Issues: Critical Assessment and Future Directions*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 103–107.

[13] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 92–101.

[14] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130.

[15] M. Caneill, D. M. Germán, and S. Zacchiroli, "The debsources dataset: two decades of free and open source software," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1405–1437, Jun 2017.

[16] G. Gousios and A. Zaidman, "A dataset for pull-based development research," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 368–371.

[17] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 422–431.

[18] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Orion: A software project search engine with integrated diverse software artifacts," in *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '13, July 2013, pp. 242–245.

[19] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proceedings of the 14th Working Conference on Reverse Engineering*, ser. WCRE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 70–79.

[20] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in postgresql," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 179–180.

[21] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, ser. WCRE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 112–.

[22] ——, "Restructuring program identifier names," in *Proceedings 2000 International Conference on Software Maintenance*, Oct 2000, pp. 97–107.

[23] E. Avidan and D. G. Feitelson, "Effects of variable names on comprehension an empirical study," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 55–65. [Online]. Available: https://doi.org/10.1109/ICPC.2017.27

[24] J. C. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," *Empirical Softw. Engg.*, vol. 24, no. 1, pp. 417–443, Feb. 2019.

[25] S. Butler, M. Wermelinger, and Y. Yu, "Investigating naming convention adherence in Java references," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 41–50.

[26] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Mining Java class naming conventions," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 93–102.

[27] ——, "Exploring the influence of identifier names on code quality: An empirical study," in *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, ser. CSMR '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 156–165.

[28] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: The Javadocminer," in *Proceedings of the Natural Language Processing and Information Systems, and 15th International Conference on Applications of Natural Language to Information Systems*, ser. NLDB'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 68–79.

[29] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1063–1073.

[30] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 38–49.

[31] A. Jaffe, J. Lacomis, E. J. Schwartz, C. L. Goues, and B. Vasilescu, "Meaningful variable names for decompiled code: A machine translation approach," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 20–30.

[32] N. Madani, L. Guerrouj, M. Di Penta, Y. Gueheneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *2010 14th European Conference on Software Maintenance and Reengineering*, March 2010, pp. 68–77.

[33] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, "Part-of-speech tagging of program identifiers for improved text-based software engineering tools," in *Proc. of the 21st International Conference on Program Comprehension (ICPC 2013)*, 2013, pp. 3–12.