Classifying code comments in Java open-source software systems

Luca Pascarella
Delft University of Technology
Delft, The Netherlands
L.Pascarella@tudelft.nl

Alberto Bacchelli
Delft University of Technology
Delft, The Netherlands
A.Bacchelli@tudelft.nl

***Abstract*—Code comments are a key software component containing information about the underlying implementation. Several studies have shown that code comments enhance the readability of the code. Nevertheless, not all the comments have the same goal and target audience. In this paper, we investigate how six diverse Java OSS projects use code comments, with the aim of understanding their purpose. Through our analysis, we produce a taxonomy of source code comments; subsequently, we investigate how often each category occur by manually classifying more than 2,000 code comments from the aforementioned projects. In addition, we conduct an initial evaluation on how to automatically classify code comments at line level into our taxonomy using machine learning; initial results are promising and suggest that an accurate classification is within reach.**

I. Introduction

While writing and reading source code, software engineers routinely introduce code comments [6]. Several researchers investigated the usefulness of these comments, showing that thoroughly commented code is more readable and maintainable. For example, Woodfield *et al.* conducted one of the first experiments demonstrating that code comments improve program readability [35]; Tenny *et al.* confirmed these results with more experiments [31], [32]. Hartzman *et al.* investigated the economical maintenance of large software products showing that comments are crucial for maintenance [12]. Jiang *et al.* found that comments that are misaligned to the annotated functions confuse authors of future code changes [13]. Overall, given these results, having abundant comments in the source code is a recognized good practice [4]. Accordingly, researchers proposed to evaluate code quality with a new metric based on code/comment ratio [21], [9].

Nevertheless, not all the comments are the same. This is evident, for example, by glancing through the comments in a source code file[1] from the Java Apache Hadoop Framework [1]. In fact, we see that some comments target end-user programmers (*e.g.*, Javadoc), while others target internal developers (*e.g.*, *inline* comments); moreover, each comment is used for a different purpose, such as providing the implementation rationale, separating logical blocks, and adding reminders; finally, the interpretation of a comment also depends on its position with respect to the source code.

Defining a taxonomy of the source code comments that developers produce is an open research problem.

[1]https://tinyurl.com/zqeqgpq

Haouari *et al.* [11] and Steidl *et al.* [28] presented the earliest and most significant results in comments' classification. Haouari *et al.* investigated developers' commenting habits, focusing on the position of comments with respect to source code and proposing an initial taxonomy that includes four high-level categories [11]; Steidl *et al.* proposed a semi-automated approach for the quantitative and qualitative evaluation of comment quality, based on classifying comments in seven high-level categories [28]. In spite of the innovative techniques they proposed to both understanding developers' commenting habits and assessing comments' quality, the classification of comments was not in their primary focus.

In this paper, we focus on increasing our empirical understanding of the types of comments that developers write in source code files. This is a key step to guide future research on the topic. Moreover, this increased understanding has the potential to (1) improve current quality analysis approaches that are restricted to the comment ratio metric only [21], [9] and to (2) strengthen the reliability of other mining approaches that use source code comments as input (*e.g.*, [30], [23]).

To this aim, we conducted an in-depth analysis of the comments in the source code files of six major OSS systems in Java. We set up our study as an exploratory investigation. We started without hypotheses regarding the content of source code comments, with the aim of discovering their purposes and roles, their format, and their frequency. To this end, we (1) conducted three iterative content analysis sessions (involving four researchers) over 50 source files including about 250 comment blocks to define an initial taxonomy of code comments, (2) validated the taxonomy externally with 3 developers, (3) inspected $2,000$ source code files and manually classified (using a new application we devised for this purpose) over $15,000$ comment blocks comprising more than $28,000$ lines, and (4) used the resulting dataset to evaluate how effectively comments can be automatically classified.

Our results show that developers write comments with a large variety of different meanings and that this should be taken into account by analyses and techniques that rely on code comments. The most prominent category of comments summarizes the purpose of the code, confirming the importance of research related to automatically creating this type of comments. Finally, our automated classification approach reaches promising initial results.

II. Motivating Example

```
1  public class STSubscriptExpression extends STExpression {
2
3      private static CSpellingService fInstance;
4
5      /**
6       * Returns the created expression, or null in case of error.
7       * @deprecated Replaced by {@link #getExpression()}
8       */
9      @Deprecated
10     public STExpression getSubscriptExpression(){
11         if (fInstance == null) {
12             fInstance = new Expression(ConsoleEditors.getPreferenceStore());
13         }
14         return fInstance;
15     }
16
17     /**
18      * Handle terminated sub-launch
19      * @param launch a terminable launch object.
20      * @author Jesse MC Wilson
21      */
22     private void STLaunchTerminated(ILaunch launch) {
23         // See com.vaadin.data.query.QueryDelegate#getPrimaryKeyColumns
24         if (this == launch)
25             return;
26         // Remove sub launch, keeping the processes of the terminated launch to
27         // show the association and to keep the console content accessible
28         if (subLaunches.remove(launch) != null) {
29             // terminate ourselves if this is the last sub launch
30             if (subLaunches.size() == 0) {
31                 // TODO: Check the possibility to exclude it
32                 //monitor.exlude();
33                 monitor.subTask("Terminated"); //$NON-NLS-1$
34                 fTerminated = true;
35                 fireTerminate();
36                 // %%%
37             }
38         }
39     }
40 }
```

Listing 1. Example of Java file.

Listing 1 shows a Java source file example containing both code and comments. In a well-documented file, comments help the reader with a number of tasks, such as understanding the code, knowing the choices and rationale of authors, and finding additional references. When developers perform software maintenance, the aforementioned tasks become mandatory steps that practitioners should attend. The fluency in performing maintenance tasks depends on the quality of both code and comments. When comments are omitted, much depends on developers' ability and code complexity; when well-written comments are present, the maintenance could be simplified.

A. Code/comment ratio to measure software maintainability

When developers want to estimate the maintainability of code, one of the easiest solutions consists in the evaluation of the code/comment ratio proposed by Garcia *et al.* [9]. By evaluating the aforementioned metric in the snippet in Listing 1, we find an overall indicator of good quality. However, the evaluated measure is inaccurate. The limitation arises from the fact that this metric considers only one kind of comment. More precisely, Garcia *et al.* focus only on the presence or absence of comments, omitting the possibility of use comments with different benefits for different end-users. Unfortunately, the previous sample of code represents a case where the author used comments for different purposes. The comment on line 31 represents a note that developers use to remember an activity, an improvement, or a fix. On line 20 the author marks his contribution on the file. Both these two comments represent real cases where the presence of comments increases the code/comment ratio without any real effect on code readability. This situation hinders the validity of this kind of metric and indicates the need for a more accurate approach to tackle the problem.

B. An existing taxonomy of source code comments

A great source of inspiration for our work comes from Steidl *et al.* who presented a first detailed approach for evaluating comment quality [28]. One of the key steps of their approach is to first automatically categorize the comments to differentiate between different comment types. They define a preliminary taxonomy of comments that comprises 7 high-level categories: Copyright, Header, Member, Inline, Section, Code, and Task. They provide evidence that their quality model, based on this taxonomy, provides important insights on documentation quality and can reveal quality defects in practice.

The study of Steidl *et al.* demonstrates the importance of treating comments in a way that suits their different categories. However, the creation of the taxonomy was not the focus of their work, as also witnessed by the few details given about the process that led to its creation. In fact, we found a number of cases in which the categories did not provide adequate information or did not differentiate the type of comments enough to obtain a clear understanding. To further clarify this, we consider three examples taken from Listing 1:

Member category. Lines 5, 6, 7 and 8 correspond to the Member category in the taxonomy by Steidl *et al.* In fact, Member comments describe the features of a method or field being located near to definition [28]. Nevertheless, we see that the function of line 6 differs from that of line 7; the former summarizes the purpose of the method, the latter gives notice about replacing the usage of the method with an alternative. By classifying these two lines together, one would lose this important difference.

IDE directives. Lines 33 does not belong to any explicit category in the taxonomy by Steidl *et al.* In this case, the target is not a developer, but another stakeholder: the Integrated Development Environment (IDE). Similarly, line 23 does not have a category, while it is a possibly important external reference to read for more details.

Noise. Line 36 represents a case of a comment that should be disregarded from any further analysis. Since it does not separate parts, the Section would not apply and an automated classification approach would try to wrongly assign it to one of the other categories. No sort of *noise* category is considered.

With our work, we specifically focus on devising an empirically grounded, fine-grained classification of comments that expands on previous initial efforts. Our aim is to get a comprehensive view of the comments, by focusing on the purpose of the comments written by developers. Besides improving our scientific understanding of this type of artifacts, we expect this work to be also beneficial, for example, to the effectiveness of the quality model proposed by Steidl *et al.* and other approaches relying on mining and analyzing code comments (*e.g.*, [21], [30], [23]).

III. METHODOLOGY

This section defines the overall goal of our study, motivates our research questions, and outlines our research method.

A. Research Questions

The ultimate goal of this study is to understand and classify the primary purpose of code comments written by software developers. In fact, past research showed evidence that comments provide practitioners with a great assistance during maintenance and future development, but not all the comments are the same or bring the same value.

We started analyzing past literature looking for similar efforts on analysis of code comments. We observed that only a few studies define a rudimentary taxonomy of comments and none of them provides an exhaustive categorization of all kinds of comments. Most of past work focuses on the impact of comments on software development processes such as code understanding, maintenance, or code review and the classification of comments is only treated as a side outcome (*e.g.*, [31], [32]). Therefore, we set our first research question:

> **RQ1.** How can code comments be categorized?

Given the importance of comments in software development, the natural next step is to apply the resulting taxonomy and investigate on the primary use of comments. Therefore, we investigate whether some classes of comments are predominant and whether there is a pattern across different projects. This investigation is reflected in our second research question:

> **RQ2.** How often does each category occur?

Finally, we investigate to what extent an automated approach can classify unseen code comments according to the taxonomy defined in RQ1. An accurate automated classification mechanism is the first essential step in using the taxonomy to mine information from large-scale projects and to improve existing approaches that rely on code comments. This leads to our last research question:

> **RQ3.** How effective is an automated approach, based on machine learning, in classifying code comments?

B. Selection of subject systems

To conduct our analysis, we focused on a single programming language (*i.e.*, Java, one of the most popular programming languages [5]) and on projects whose source code is publicly available, *i.e.*, open-source software (OSS) projects. Particularly, we selected six heterogeneous software systems: Apache Spark [2], Eclipse CDT, Google Guava, Apache Hadoop, Google Guice, and Vaadin. They are all open source projects and the history of the changes are controlled with GIT version control system. Table I details the selected systems. We select unrelated projects emerging from the context of different four software ecosystems (*i.e.*, Apache, Google, Eclipse, and Vaadin); the development environment, the number of contributors, and the project size are different, thus mitigating some threats to the external validity.

Table I
DETAILS OF THE SUBJECT OSS SYSTEMS

Project	Java source lines			Commits	Contributors	Sample sets	
	Code	Comment	Ratio			Files	Blocks of comments
Apache Spark	753k	287k	38%	38k	1,351	61	465
Eclipse CDT	1,239k	466k	38%	26k	211	799	6,009
Google Guava	252k	88k	35%	4k	185	158	1,100
Apache Hadoop	1,258k	396k	31%	15k	171	672	4,228
Google Guice	9k	5k	56%	2k	32	59	718
Vaadin	2,643k	1,101k	42%	91k	726	401	3,340

C. Categorization of code comments

To answer our first research question about categorizing code comments, we conducted three iterative *content analysis sessions* [15] involving 4 software engineering researchers (3 Ph.D. candidates and 1 faculty member) with at least 3 years of programming experience. Two of these researchers are authors of this paper. In the first iteration, we started choosing 6 appropriate projects (reported in Table I) and sampling 35 files with a large variety of code comments. Subsequently, together we analyzed all source code and comments. During this analysis we could define some obvious categories and left undecided some comments; this resulted in the first draft taxonomy defining temporary category names. In the course of the second phase, we first conducted an individual work analyzing 10 new files, in order to check or suggest improvements to the previous taxonomy, then we gathered together to discuss the findings. The second phase resulted in a validation of some clusters in our draft and the redefinitions of others. The third phase was conducted in team and we analyzed 5 files that were previously unseen. During this session we completed the final draft of our taxonomy verifying that each kind of comments we encountered was covered by our definitions and those overlapping categories were absent.

Through this iterative process, we defined a taxonomy having a hierarchy with two layers. The top layer consists of 6 categories and the inner layer consists of 16 subcategories.

Validation. We externally validated the resulting taxonomy with 3 professional developers having 3 to 5 years of Java programming experience. We conducted one session with each developer. At the beginning of the session, the developer received a printed copy of the description of the comment categories in our taxonomy (similar to the explanation we provide in Section IV-A) and was allowed to read through it and ask questions to the researcher guiding the session. Afterwards, the developer was required to login into COMMEAN (a web application, described in Section III-D, that we devised for this task and to facilitate the large-scale manual classification necessary to answer RQ2 and RQ3) and classify each comment in 3 Java source code files (the same files have been used for all the developers), according to the provided taxonomy. During the classification, the researcher was not in the experiment room, but the printed taxonomy could be

consulted. At the end of the session, the guiding researcher came back to the experiment room and asked the participant to comment on the taxonomy and the classification task. At the end of all three sessions, we compared the differences (if any) among the classifications that the developers produced.

All the participants found the categories clear and the task feasible; however, they also reported the need for consulting the printed taxonomy several times during the session to make sure that their choice was in line with the description of the category. The analysis of the three sets of answers showed a few minor differences with an agreement above 92%. The differences were all within the same top category and mostly regarding where the developers split certain code blocks into two sub-categories.

D. A dataset of categorized code comments, publicly available

To answer the second research question about the frequencies of each category, we needed a statistically significant set of code comments classified accordingly to the taxonomy produced as an answer to RQ1.

Sampling approach. Since the classification had to be done manually, we relied on random sampling to produce a statistically significant set of code comments from each one of the six OSS projects we considered in our study. To establish the size of such sample sets, we used as a unit the number of files, rather than number of comments: This results in sample sets that give a more realistic overview of how comments are distributed in a system. In particular, we established the size (n) of such set with the following formula [33]:

$$n = \frac{N \cdot \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}{(N-1)\,E^2 + \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}$$

The size has been chosen to allow the simple random sampling without replacement. In the formula $\hat{p}$ is a value between 0 and 1 that represents the proportion of files containing a specific block of code comment, while $\hat{q}$ is the proportion of files not containing such kind of comment. Since the *a-priori* proportion of $\hat{p}$ is not known, we consider the worst case scenario where $\hat{p} \cdot \hat{q} = 0.25$. In addition, considering we are dealing with a small population (*i.e.*, 557 Java files for Google Guice project) we use the finite population correction factor to take into account their size (N). We sample to reach a confidence level of 95% and error (E) of 5% (*i.e.*, if a specific comment entity is present in $f\%$ of the files in the sample set, we are 95% confident it will be in $f\% \pm 5\%$ files of our population). The suggested value for the sample set is $1,925$ files. In addition, since we split the sample sets in two parts with an overlapped chunk for validation, we finally sampled $2,000$ files. This value does not change significantly the error level that remains close to 5%. This choice only validates the quality of our dataset as a representation of the overall population: It is not related to the *precision* and *recall* values presented later, which are actual values based on manually analyzed elements.

Manual classification. Once the sample of files with comments was selected, each of them had to be manually classified according to our taxonomy. To facilitate this error-prone and time-consuming task, we build a web application, named CommMean. Figure 1 shows the main page of CommMean, which comprises the following components:

- The *Actions* panel (1) handles the authentication of the users and several actions such as 'start', 'suspend', or 'send classification'. In addition, the panel keeps the user updated on the status of the classification showing the path of the resource loaded in the application and the progress with the following syntax: I-P/T. Where I represents the current index, P is the progress, and T is the total number of files to be processed.
- The *Annotation* panel (2) allows the user to append a pre-defined label to the selected text or define a new label. It enables the possibility to append a free text comment, create a link between comments and code, or categorize text composed of multiple parts. In addition, two keyboard shortcuts help the user to append the current label to selected text and create a link between source code and comments.
- The *Source view* panel (3) is the main view of the application. It contains the Java source file with highlighted syntax to help users during the classification and increase the quality of the analysis. In addition, the processed parts of the file are marked with different colors.
- The *Status* panel (4) shows the progress of the current file. A dynamic table is created when a new comment is added. A row of the table contains the initial position, the final position, the label used in the categorization, a summary of how many parts compose it, and a summary of linked code (if any). Clicking on rows, the correspondent text is highlighted and using the delete button the user is able to cancel a wrong classification.
- The *Selection* panel (5) shows details such as selected test, initial position, final position, and length of the text.

The two authors of this paper manually inspected the sample set composed of $2,000$ files. One author analyzed 100% of these files, while another analyzed a random, overlapping subset comprising 10% of the files. These overlapped files were used to verify their agreement, which, similarly to the external validation of the taxonomy with professional developers (Section III-C), highlighted only negligible differences. Moreover, this large-scale categorization also confirmed the exhaustiveness of the taxonomy created in RQ1: None of the annotators felt that comments, or parts of the comments, should have been classified by creating a new category.

Finally, the two researchers annotated, when present, any link between comments and the code they are referring to. This allows the use of our dataset for future approaches that attempt to recover the traceability links between code and comments. We make our dataset publicly available [24].

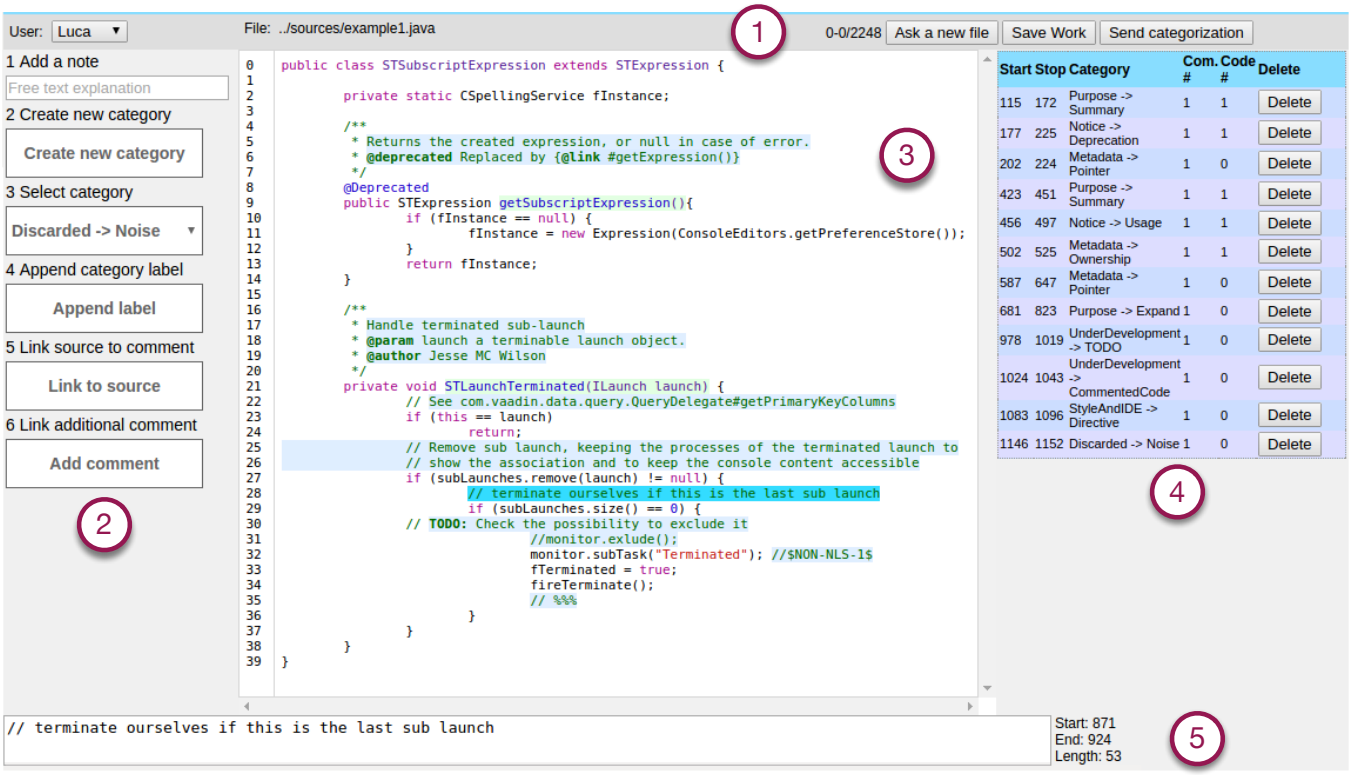

Figure 1. COMMEAN: our web application for classifying code comments content.

E. Automated classification of source code comments

In the third research question we set to investigate to what extent and with which accuracy source code comments can be automatically categorized according to the taxonomy resulting from the answer to RQ1. Employing sophisticated classification techniques (*e.g.*, based on deep learning approaches [10]) to accomplish this task goes beyond the scope of the current work. Our aim is to two-fold: (1) Verifying whether it is feasible to create an automatic classification approach that provides fair accuracy and (2) defining a reasonable baseline against which future methods that aim at a more accurate, project-specific classification can be tested.

Classification granularity. We set the automated classification to work *at line level*. In fact, from our manual classification, we found several blocks of comments that had to be split and classified into different categories (similarly to the block defined in the lines 5–8 in Listing 1) and in the vast majority of the cases (96%), the split was at line level. In only less than 4% of the cases, one line had to be classified into more than one category. In these cases, we replicated the line in our dataset for each of the assigned categories, to get a lower bound on the effectiveness in these cases.

Classification technique. Having created a reasonably large dataset to answer RQ2 (it comprises more than 15,000 comment blocks totaling over 30,000 lines), we employ *supervised* machine learning [8] to build the automated classification approach. This kind of machine learning uses a pre-classified set of samples to infer the classification function. In particular, we tested two different classes of supervised classifiers: (1) *probabilistic classifiers*, such as naive Bayes or naive Bayes Multinominal, and (2) *decision tree algorithms*, such as J48 and Random Forest. These classes make different assumptions on the underlying data, as well as have different advantages and drawbacks in terms of execution speed and overfitting.

Classification evaluation. To evaluate the effectiveness of our automated technique to classification code comments into our taxonomy, we measured two well known Information Retrieval (IR) metrics for the quality of results [18], named *precision* and *recall*:

$$Precision = \frac{|TP|}{|TP + FP|}$$

$$Recall = \frac{|TP|}{|TP + FN|}$$

TP, FP, and FN are based on the following definitions:

- TRUE POSITIVES (TP): elements that are correctly retrieved by the approach under analysis (*i.e.*, comments categorized in accord to annotators)
- FALSE POSITIVES (FP): elements that are wrongly classified by the approach under analysis (*i.e.*, comments categorized in a different way by the oracle)
- FALSE NEGATIVES (FN): elements that are not retrieved by the approach under analysis (*i.e.*, comments present only in the oracle)

The union of TP and FN constitutes the set of correct classifications for a given category (or overall) present in the benchmark, while the union of TP and FP constitutes the set of comments as classified by the used approach. In other words, *precision* represents the fraction of the comments that are correctly classified into a given category, while *recall* represents the fraction of correct comments in that category.

F. Threats to validity

Sample validity. One potential criticism of a scientific study conducted on a small sample of projects is that it could deliver little knowledge. In addition, the study highlights the characteristics and distributions of 6 open source frameworks mainly focusing on developers practices rather than end-users patterns. Historical evidence shows otherwise: Flyvbjerg gave many examples of individual cases contributing to discoveries in physics, economics, and social science [7]. To answer to our research questions, we read and inspected more than $28,000$ lines of comments belonging to $2,000$ Java files (see Section III-D) written by more than $3,000$ contributors in 6 different projects (in accord to Table I). We also chose projects belonging to four open-source software ecosystems and with different development environments, number of contributors, and size of the project.

Taxonomy validity. To ensure that the comments categories emerged from our content analysis sessions were clear and accurate, and to evaluate whether our taxonomy provides an exhaustive and effective way to organize source code comments, we conduced a validation session that involved three experienced developers (see Section III-C) external to content analysis sessions. These software engineers conducted an individual session on 3 unrelated Java source files. They observed that categories were clear and the task feasible, and the analysis of the three sets of answers showed a few minor differences with an agreement above 92%. In addition, we reduce the impact of human errors during the creation of the dataset by developing CODEMEAN, a web application to assist the annotation process.

External validity. Threats come with the generalization of our results. The proposed approach may show different result on different target systems. To reduce this limitation we selected 6 projects with unrelated characteristics and with different size in term of contributors and number of lines. To judge the generalizability we tested our results simulating this circumstance using the project cross validation. Similarly, another threat concerning the generalizability is that our taxonomy refers only to a single object-oriented programming language *i.e.*, Java. However, since many object-oriented languages descend to common ancestor languages, many functionalities across object-oriented programming are similar and it is reasonable to expect the same to happen for their corresponding comments. Further research can be designed to investigate whether our results hold in other programming paradigms.

IV. Results and Analysis

In this section, we present and analyze the results of our research questions aimed at understanding what developers write in comments and with which frequency, as well as at evaluating the results of an automated classification approach.

A. RQ1. How can code comments be categorized?

Our manual analysis led to the creation of a taxonomy of comments having a hierarchy with two layers. The top level categories gather comments with similar overall purpose, the internal levels provide a fine-grained definition using explanatory names. The top level categories are composed of 6 distinct groups and the second level categories are composed of 16 definitions. We now describe each category with the corresponding subcategories.

A. PURPOSE

The PURPOSE category contains the code comments used to describe the functionality of linked source code either in a shorter way than the code itself or in a more exhaustive manner. Moreover, these comments are often written in a pure natural language and are used to describe the purpose or the behavior of the referenced source code. The keywords *'what'*, *'how'* and *'why'* describe the actions that take place in the source code in SUMMARY, EXPAND, and RATIONALE groups, respectively, which are the subcategories of PURPOSE:

A.1 **SUMMARY:** This type of comments contains a brief description of the behavior of the source code referenced. To highlight this type of comments the question word *'what'* is used. Intuitively, this category incorporates comments that represent a sharp description of what the code does. Often, this kind of comments is used by developers to provide a summary that helps understanding the behavior of the code without reading it.

A.2 **EXPAND:** As with the previous category, the main purpose of reading this type of comment is to obtain a description of the associated code. In this case, the goal is to provide *more details* on the code itself. The question word *'how'* can be used to easily recognize the comments belonging to this category. Usually, these comments explain in detail the purpose of short parts of the code, such as details about a variable declaration.

A.3 **RATIONALE:** This type of comment is used to explain the rationale behind some choices, patterns, or options. The comments that answer the question *'why'* belong to that category (*e.g.*, "Why does the code use that implementation?" or "Why did the developer use this specific option?").

B. NOTICE

The NOTICE category contains the comments related to the description of warning, alerts, messages, or in general, functionalities that should be used with care. It also includes the reasons and the explanation of some developers' choices. In addition, it covers the description of the adopted strategies to

solve a bug, improve performance, prevent fault, etc. Further, it covers the use case examples giving to developer additional advice over parameters or options. Moreover, it covers examples of use cases or warnings about exceptions.

B.1 **DEPRECATION:** This type of comments contains explicit warnings used to inform the users about deprecated interface artifacts. This subcategory contains comments related to alternative methods or classes that should be used (*e.g.*, "do not use [this]", "is it safe to use?" or "refer to: [ref]"). It also includes the description of the future or scheduled deprecation to inform the users of candidate changes. Sometimes, a tag comment such as *@version*, *@deprecated*, or *@since* is used.

B.2 **USAGE:** This type of comments regards explicit suggestions to users that are planning to use a functionality. It combines pure natural language text with examples, use cases, snippets of code, etc. Often, the advice is preceded by a metadata mark *e.g.*, *@usage*, *@param* or *@return*

B.3 **EXCEPTION:** This category describes the reasons for the occurred exception. Sometimes it contains potential suggestions to prevent the unwanted behavior or actions to do when that event arise. Some tags are used also in this case, such as *@throws* and *@exception*.

C. **UNDER DEVELOPMENT**

The UNDER DEVELOPMENT category covers the topics related to ongoing and future development. In addition, it envelopes temporary tips, notes, or suggestions that developers use during development. Sometimes informal requests of improvement or bug correction may also appear.

C.1 **TODO:** This type of comments regards explicit actions to be done or remarks both for the owners of the file and for other developers. It contains explicit fix notes about bugs to analyze and resolve, or already treated and fixed. Furthermore, it references to implicit TODO actions that may be potential enhancements or fixes.

C.2 **INCOMPLETE:** This type comprises partial, pending or empty comment bodies. It may be introduced intentionally or accidentally by developers and left in the incomplete state for some reason. This type may be added automatically by the IDE and not filled in by the developer *e.g.*, empty "@param" or "@return" directives.

C.3 **COMMENTED CODE:** This category is composed of comments that contain source code commented out by developers. It envelopes functional code in a comment to try hidden features or some work in progress. Usually, this type of comments represents features under test or temporarily removed. The effect of this kind of comments is directly transposed on the program flow.

D. **STYLE & IDE**

The STYLE & IDE category contains comments that are used to logically separate the code or provide special services. These comments may be added automatically by the IDE or used to communicate with it.

D.1 **DIRECTIVE:** This is an additional text used to communicate with the IDE. It is in form of comments to be easily skipped by the compiler and it contains text of limited meaning to human readers. These comments are often added automatically by the IDE or used by developers to change the default behavior of the IDE or compiler.

D.2 **FORMATTER:** This type of comments represents a simple solution adopted by the developers to separate the source code in logical sections. The occurrence of patterns or the repetition of symbols is a good hint at the presence of a comment in the formatter category.

E. **METADATA**

The METADATA category aims to classify comments that define meta information about the code, such as authors, license, and external references. Usually, some specific tags (*e.g.*, *"@author"*) are used to mark the developer name and its ownership. The license section provides the legal information about the source code rights or the intellectual property.

E.1 **LICENSE:** Generally placed on top of the file, this types of comments describes the end-user license agreement, the terms of use, the possibility to study, share and modify the related resource. Commonly, it contains only a preliminary description and some external references to the complete policy agreement.

E.2 **OWNERSHIP:** These comments describe the authors and the ownership with different granularity. They may address methods, classes or files. In addition, this type of comments includes credentials or external references about the developers. A special tag is often used *e.g.*, *"@author"*.

E.3 **POINTER:** This types of comments contains references to linked resources. The common tags are: *"@see"*, *"@link"* and *"@url"*. Other times developers use custom references such as *"FIX #2611"* or *"BUG #82100"* that are examples of traditional external resources.

F. **DISCARDED**

This category groups the comments that do not fit into the categories previously defined; they have two flavors:

F.1 **AUTOMATICALLY GENERATED:** This category defines auto-generated notes (*e.g.*, "Auto-generated method stub"). In most case, the comment represents the skeleton with a comment's placeholder provided by the IDE and left untouched by the developers.

F.2 **NOISE:** This category contains all remaining comments that are not covered by the previous categories. In addition, it contains the comments whose meaning is hard to understand due to their poor content (*e.g.*, meaningless because out of context).

B. RQ2. How often does each category occur?

The second research question investigates the occurrence of each category of comments in the $2,000$ source files that we manually classified from our 6 OSS subject projects.

Figure 2. Frequencies of comments per category. Top, red bars show the occurrences by blocks of comments and bottom, blue bars by lines.

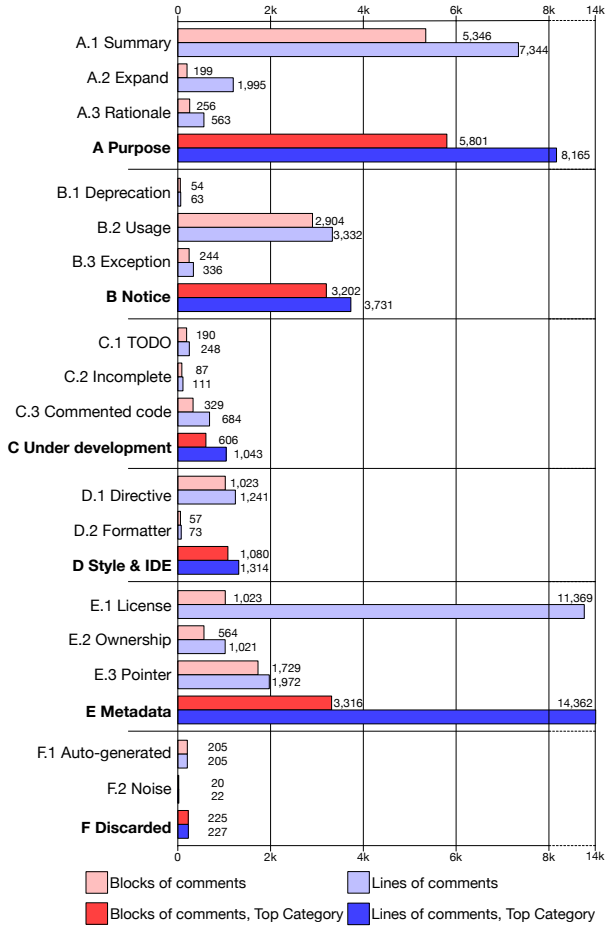

Figure 2 shows the distribution of the comments across the categories; it reports the cumulative value for the top level categories (*e.g.*, NOTICE) and the absolute value for the inner categories (*e.g.*, EXCEPTION). For each category, the top red bar indicates the number of *blocks of comments* in the category, while the bottom blue bar indicates the number of non-blank *lines of comments* in the category.

Comparing blocks and lines, we see that, unsurprisingly, the longest type of comments is LICENSE, with more than 11 lines on average per block. The EXPAND category follows with a similar average length. The SUMMARY category has only an average length of 1.4 lines, which is surprising, since it is used to describe the purpose of possibly very long methods, variables, or blocks of code. The remaining categories show negligible differences between number of blocks and lines.

We consider the quality metric code/comment ratio, which was proposed at line granularity [21], [9], in the light of our results. We see that 59% of lines of comments should not be considered (*i.e.*, categories from C to F), as they do not reflect any aspect of the readability and maintainability of the code they pertain to; this would significantly change the results. On the other hand, if one considers blocks of comments, the result would be closer to the aspect that is set to measure with the code/comment metric. In this case, a simple solution would be to only filter out the METADATA category, because the other categories seem to have a more negligible impact.

Considering the distribution of the comments, we see that the SUMMARY subcategory is the most prominent one. This confirms the value of research efforts that attempt to generate summaries for functions and methods automatically, by analyzing the source code [26]. In fact, these methods would alleviate developers from the burden of writing a significant amount of the comments we found in source code files. On the other hand, the SUMMARY accounts for only 24% of the overall lines of comments, thus suggesting that they only give a partial picture on the variety and role of this type of documentation. The second most prominent category is USAGE. Together with the prominence of SUMMARY, this suggests that the comments in the systems we analyzed are targeting end-user developers more frequently than internal developers. This is also confirmed by the low occurrence of the UNDER DEVELOPMENT category. Concerning UNDER DEVELOPMENT, the low number of comments in this category may also indicate that developers favor other channels to keep track of tasks to be done in the code.

Finally, the variety of categories of comments and their distribution underlines once more the importance of a classification effort before applying any analysis technique on the content and value of code comments. The low number of discarded cases corroborates the completeness of the taxonomy proposed in RQ1.

C. RQ3. How effective is an automated approach, based on machine learning, in classifying code comments?

To evaluate the effectiveness of machine learning algorithm in classifying code comments we employed a supervised learning method. Supervised machine learning bases the decision evaluating on a pre-defined set of features. Since we set to classify *lines* of code comments, we computed the features at line granularity.

Text preprocessing. We preprocessed the comments by doing the following actions in this order: (1) tokenizing the words on spaces and punctuation (except for words such as '@usage' that would remain compounded), (2) splitting identifiers based on camel-casing (*e.g.*, 'ModelTree' became 'Model Tree'), (3) lowercasing the resulting terms, (4) removing numbers and rare symbols, and (5) creating one instance per line.

Feature creation. Table II shows some of the features we devised and all that appear in the final model. Due to the optimal set of features is not known *a priori*, we started with some simple, traditional features and iteratively experimented with others more sophisticated, in order to improve precision and recall for all the projects we analyzed.

A set of features commonly used in text recognition [25] consists in measuring the occurrence of the words; in fact, words are the fundamental tokens of all languages we want to classify. To avoid overfitting to words too specific to a project, such as code identifiers, we considered only words above a certain threshold t. This value has been found experimentally,

Table II
MACHINE LEARNING FEATURES FOR COMMENTS CLASSIFICATION

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 *e.g.*, object.method(par1, par2);
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
row position	numeric	detects the absolute position of the comment
adjacent rows	numeric	recognizes the nature of the adjacent rows *e.g.*, comments or code
deprecation	boolean	true if comment contains special tags like @deprecation
usage	boolean	true if comment contains special tags such as @usage, @return or @value
exception	boolean	true if comment contains special tags such as @exception or @throws
TODO	boolean	true if comment contains keywords such as todo or fix or a link to a bug is detected
incomplete	boolean	true if comment contains an empty body
commented code	boolean	true if comment contains code snippets
directive	boolean	true if comment contains special sequence of symbols used by IDE
formatter	boolean	true if comment is composed of patterns of symbols or characters
license	boolean	true if comment contains words such as license, copyright, legal or law
ownership	boolean	true if comment contains tags such as @author or @owner
pointer	boolean	true if comment contains a reference to an external linkable resource
automatic generated	boolean	true if comment contains text automatically inserted by IDE *e.g.*, Auto-generated method stub

we started with a minimum of 3 increasing up to 10. Since the values around 7 do not change the precision and recall quality, we chose that threshold.
In addition, other features consider the information about the *context* of the line, such as the text length, the comment position in the whole file, the number of rows, the nature of the adjacent rows, *etc.*
The last set of features is category specific. We defined regular expressions to recognize specific patterns. We report three detailed examples:

- This regular expression is used to match comments in single line or multiple lines with empty body.
  `^\s*\/(\*|\s)*(\/|\*\s*\*\/)\n*$`
- This regular expression matches the special keywords used in the *Usage* category.
  `(?i)@param|@usage|@since|@value|@return`
- The following regular expression is used to find patterns of symbols that may be used in *Formatter* category.
  `([^*\s])(\1\1)|^\s*\/\/\/\s*\S*`
  `|\$\S*\s*\S*\$`

Machine learning validation with 10-fold. We tested both probabilistic classifiers and decision tree algorithms. When using probabilistic classifiers, the average values of precision and recall were usually lower than values obtained using decision tree algorithms, thus a minor number of comments are correctly classified. Conversely, using decision tree algorithm the percentage value associated with the correctly classified instances is better, with Random Forest we obtain up to 98.4% and the effect is that more comments are correctly classified. Nevertheless, in the latter case, many comments belonging to classes with a low occurrence were wrongly classified. Since the purpose of our tool is to best fit the aforementioned taxonomy we discovered that the best classifier is based on a probabilistic approach.
In Table III we report only the results (precision, recall, and weighted average TP rate) for the naive Bayes Multinominal classifier that on average, considering whole categories, achieves a better result accordingly to the aforementioned considerations. In Table III we intentionally leave empty cells that correspond to categories of comments that are not present in related projects. For the evaluation, we started with a standard 10-fold cross validation. Table III shows the results in the column '10-fold'.

Cross-project validation. Different systems have comments describing different code artifacts and are likely to use different words and jargons. Thus, term-features working for the comments in one system may not work for others. To better test the generalizability of the results achieved by the classifier, we conduct a *cross-project validation*, as also previously proposed and tested by Bacchelli *et al.* [3]. In practice, cross-project validation consists in a 6-folds cross validation, in which folds are neither stratified nor randomly taken, but correspond exactly to the different systems: We train the classifiers on 5 systems and we try to predict the classification of the comments in the remaining system. We do this six times rotating the test system. The right-most columns (*i.e.*, 'cross-project') in Table III show the results by tested system.

Summary. The values for 10-fold cross validation reported in Table III show accurate results (mostly above 0.95%) achieved for top-level categories. This means that the classifier could be used as an input for tools that analyze source code comments of the considered systems. For inner-categories, the results are lower; nevertheless, the weighted average TP rate remains 0.85. Furthermore, we do not see large effects due to the prominent class imbalance. This suggests that the amount of training data is enough for each class.
As expected, testing with cross-project validation, the classifier performance drops. However, this is a more reliable test for what to expect with JAVA comments from unseen projects. The weighted average TP rate that goes as low as 0.74. This indicates that project-specific terms are key for the classification and either an approach should start with some supervised data or more sophisticated features must be devised.

Table III
RESULTS OF THE CLASSIFICATION WITH NAIVE BAYES MULTINOMIAL CLASSIFIER

	P = Precision R = Recall		Validation						
				Cross project					
Top categories	Inner categories		10-fold	CDT	Guava	Guice	Hadoop	Vaadin	Spark
Purpose	Summary	P	0.88	0.96	0.68	0.61	0.72	0.62	0.65
		R	0.82	0.99	0.61	0.69	0.56	0.69	0.84
	Expand	P	1.00	0.84	0.00	0.00	0.09	0.00	0.00
		R	0.98	0.64	0.00	0.00	0.05	0.00	0.00
	Rational	P	0.50	0.56	0.15	0.00	0.10	0.03	0.67
		R	0.69	0.84	0.23	0.00	0.41	0.17	0.17
	Purpose	P	0.99	0.77	0.77	0.81	0.80	0.83	0.68
		R	0.99	0.98	0.98	0.81	1.00	1.00	1.00
Notice	Deprecation	P	0.74	0.75	0.22			0.14	
		R	0.78	0.81	1.00			1.00	
	Usage	P	0.86	0.85	0.50	0.43	0.67	0.90	0.56
		R	0.90	0.87	0.45	0.64	0.61	0.65	0.15
	Exception	P	0.76	0.75	0.43	0.00	0.58	0.69	0.13
		R	0.98	0.95	0.87	0.00	0.88	0.97	0.29
	Notice	P	1.00	0.50	0.50	0.36	0.60	1.00	1.00
		R	0.98	0.50	0.50	1.00	0.41	0.33	0.17
Under dev.	TODO	P	0.61	0.97	0.57	0.29	0.03	0.19	
		R	0.52	0.96	0.83	1.00	0.16	0.11	
	Incomplete	P	0.91	0.92			0.11	0.95	
		R	0.96	1.00			0.88	0.91	
	Commented code	P	0.91	0.91			0.05	0.92	
		R	0.91	0.95			0.06	0.50	
	Under development	P	0.98	1.00	0.00	0.00	0.00	0.00	
		R	0.93	0.67	0.00	0.00	0.00	0.00	
Style & IDE	Directive	P	0.96	0.96				0.00	
		R	1.00	1.00				0.00	
	Formatter	P	0.81	0.93	0.00			0.00	
		R	0.77	0.28	0.00			0.00	
	Style & IDE	P	0.97	1.00	1.00			0.00	
		R	0.99	1.00	1.00			0.00	
Metadata	License	P	0.99	1.00	0.98	1.00	0.99	0.99	1.00
		R	0.98	0.99	1.00	0.95	0.99	1.00	1.00
	Ownership	P	0.80	1.00	1.00	0.57	0.00	1.00	
		R	0.96	1.00	0.08	0.27	0.00	0.98	
	Pointer	P	0.84	0.80	0.82	0.81	0.79	0.97	1.00
		R	0.94	0.74	0.52	0.54	0.70	0.85	0.60
	Metadata	P	1.00	1.00	1.00	1.00	1.00	1.00	0.89
		R	1.00	0.68	0.68	0.57	0.57	0.95	1.00
Discarded	Auto generated	P	0.90	0.91			0.13	0.84	
		R	1.00	1.00			1.00	1.00	
	Noise	P	0.65	1.00		0.00	0.00	0.00	
		R	0.77	0.39		0.00	0.00	0.00	
	Discarded	P	0.96	0.00		0.00	0.00	0.00	
		R	0.98	0.00		0.00	0.00	0.00	
Weighted average TP rate			0.85	0.88	0.77	0.79	0.74	0.80	0.83

V. RELATED WORK

A. Information Retrieval Technique

Lawrie et al. [14] use information retrieval techniques based on cosine similarity in vector space models to assess program quality under the hypothesis that *"if the code is high quality, then the comments give a good description of the code"*. Marcus *et al.* propose a novel information retrieval techniques to automatically identify traceability links between code and documentation [19]. Similarly, de Lucia *et al.* focus on the problem of recovering traceability links between the source code and connected free text documentation. They propose a comparison between a probabilistic information retrieval model and a vector space information retrieval [16]. Even though comments are part of software documentation, previous studies on information retrieval focus generally on the relation between code and free text documentation.

B. Comments Classification

Several studies regarding code comments in the 80's and 90's concern the benefit of using comments for program comprehension [35], [31], [32]. Stamelos *et al.* suggest a simple ratio metric between code and comments, with the weak hypothesis that software quality grows if the code is more commented [27]. Similarly, other two authors define metrics for measuring the maintainability of a software system and discuss how those metrics can be combined to control quality characteristics in an efficient manner [21], [9].

New recent studies add more emphasis to the code comments in a software project. Fluri *et al.* present a heuristic approach to associate comments with code investigating whether developers comment their code. Marcus and Maletic propose an approach based on information retrieval technique [20]. Maalej and Robillard investigate API reference documentation (such as javadoc) in Java SDK 6 and .NET 4.0 proposing a taxonomy of knowledge types. They use a combination of grounded and analytical approaches to create such taxonomy. [17]. Instead Witte *et al.* used Semantic Web Technologies to connect software code and documentation artifacts [34]. However, both approaches focus on external documentation and do not investigate evolutionary aspects or quality relationship between code and comments, *i.e.*, they do not track how documentation and source code changes together over time or the combined quality factor. More in focus is the work of Steidl *et al.* where they investigate over the quality of the source code comments [29]. They proposed model for comment quality based on different comment categories and use a classification based on machine learning technique tested on Java and C/C++ programs. Despite the quality of the work, they found only 7 high-level categories of comments based mostly on comment syntax, *i.e.*, inline comments, section separator comments, task comments, *etc.* A different approach is adopted by Padioleau *et al.* [22]. The innovative idea is to create a taxonomy based on the comment's meaning. Even if it is more difficult to extract the content from human sentences, their proposal is a more suitable technique for defining a taxonomy. We follow this path in our work.

VI. CONCLUSION

Code comments contain valuable information to support software development especially during code reading and code maintenance. Nevertheless, not all the comments are the same, for accurate investigations, analyses, usages, and mining of code comments, this has to be taken into account. In this work we investigated how comments can be categorized, also proposing an approach for their automatic classification.

The contributions of our work are:

- A novel, empirically validated, hierarchical taxonomy of code comments for Java projects, comprising 16 inner categories and 6 top categories.
- An assessment of the relative frequency of comment categories in 6 OSS Java software systems.
- A publicly available dataset of more than 2,000 source code files with manually classified comments, also linked to the source code entities they refer to.
- An empirical evaluation of a machine learning approach to automatically classify code comments according to the aforementioned taxonomy.

REFERENCES

[1] Apache Software Foundation (ASF) - Apache Hadoop software library. http://hadoop.apache.org/. [Online; accessed 10-02-2017].

[2] Apache Spark. http://spark.apache.org. [Online; accessed 03-Feb-2016].

[3] A. Bacchelli, T. dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *In Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, pages 375–385, 2012.

[4] 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 & designing for pervasive information*, pages 68–75. ACM, 2005.

[5] N. Diakopoulos and S. Cass. The top programming languages 2016. *IEEE Spectrum*, http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016, Jul 2016.

[6] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 70–79. IEEE, 2007.

[7] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.

[8] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[9] M. J. B. Garcia and J. C. Granja-Alvarez. Maintainability as a key factor in maintenance productivity: a case study. In *icsm*, page 87, 1996.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. The MIT Press, 2016.

[11] D. Haouari, H. Sahraoui, and P. Langlais. How good is your comment? a study of comments in java programs. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 137–146. IEEE, 2011.

[12] C. S. Hartzman and C. F. Austin. Maintenance productivity: Observations based on an experience in a large system environment. pages 138–170, 1993.

[13] 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*, MSR '06, pages 179–180, New York, NY, USA, 2006. ACM.

[14] D. J. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 149–158. IEEE, 2006.

[15] W. Lidwell, K. Holden, and J. Butler. *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design*. Rockport Publishers, 2nd edition, January 2010.

[16] D. Lucia et al. Information retrieval models for recovering traceability links between code and documentation. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 40–49. IEEE, 2000.

[17] W. Maalej and M. P. Robillard. Patterns of knowledge in api reference documentation. *IEEE Trans. Softw. Eng.*, 39(9):1264–1282, Sept. 2013.

[18] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[19] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135. IEEE, 2003.

[20] A. Marcus, J. I. Maletic, and A. Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(5):811–836, 2005.

[21] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[22] Y. Padioleau, L. Tan, and Y. Zhou. Listening to programmers - taxonomies and characteristics of comments in operating system code. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 331–341, 2009.

[23] Y. Padioleau, L. Tan, and Y. Zhou. Listening to programmers taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*, pages 331–341. IEEE Computer Society, 2009.

[24] L. Pascarella and A. Bacchelli. Manually classified dataset of source code comments. http://doi.org/10.4121/uuid:232d15bf-ce75-48f5-8a2c-e8e809b8333e.

[25] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[26] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[27] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[28] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 83–92. IEEE, 2013.

[29] D. Steidl, B. Hummel, and E. Jürgens. Quality analysis of source code comments. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, pages 83–92, 2013.

[30] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.

[31] T. Tenny. Procedures and comments vs. the banker's algorithm. *SIGCSE Bull.*, 17(3):44–53, Sept. 1985.

[32] T. Tenny. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, 1988.

[33] M. F. Triola. *Elementary statistics*. Pearson/Addison-Wesley Reading, MA, 2006.

[34] R. Witte, Y. Zhang, and J. Rilling. Empowering software maintainers with semantic web technologies. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*, pages 37–52, 2007.

[35] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. pages 215–223, 1981.