Quality Analysis of Source Code Comments

Daniela Steidl Benjamin Hummel Elmar Juergens
CQSE GmbH, Garching b. München, Germany
{steidl,hummel,juergens}@cqse.eu

***Abstract*—A significant amount of source code in software systems consists of comments, *i. e.*, parts of the code which are ignored by the compiler. Comments in code represent a main source for system documentation and are hence key for source code understanding with respect to development and maintenance. Although many software developers consider comments to be crucial for program understanding, existing approaches for software quality analysis ignore system commenting or make only quantitative claims. Hence, current quality analyzes do not take a significant part of the software into account. In this work, we present a first detailed approach for quality analysis and assessment of code comments. The approach provides a model for comment quality which is based on different comment categories. To categorize comments, we use machine learning on Java and C/C++ programs. The model comprises different quality aspects: by providing metrics tailored to suit specific categories, we show how quality aspects of the model can be assessed. The validity of the metrics is evaluated with a survey among 16 experienced software developers, a case study demonstrates the relevance of the metrics in practice.**

I. Introduction

A significant amount of source code in software systems consists of comments, which document the implementation and help developers to understand the code, *e. g.*, for later modification or reuse: Several researchers have conducted experiments showing that commented code is easier to understand than code without comments [1], [2]. Comments are the second most-used documentary artifact for code understanding, behind only the code itself [3]. In addition, source code documentation is also vital in maintenance and forms an important part of the general documentation of a system. In contrast to external documentation, comments in source code are a convenient way for developers to keep documentation and code consistently up to date. Developers widely agree that poor general documentation leads to misunderstandings [4], and studies have shown that poor documentation significantly lowers the maintainability of software [5]. Although developers commonly agree on the importance of software documentation [3], commenting code is often neglected due to release deadlines and other time pressure during development.

Previous approaches to analyzing software quality ignore comments, or make only quantitative claims: they evaluate the comment ratio of a system to measure documentation quality [6], [7]. However, this metric is not sufficient: Several comments (copyrights or commented out code) should be excluded as they do not enhance system understanding and quantitative measures cannot detect outdated/ useless comments.

Furthermore, a complete model of comment quality does not exist. Coding conventions, *e. g.*, marginally touch on the topic of commenting code but mostly lack depth and precision [8]. So far, (semi-) automatic methods for comment quality assessment have not been developed as comment analysis is a difficult task: Comments comprise natural language and have no mandatory format aside from syntactic delimiters. Hence, algorithmic solutions will be heuristic in nature.

Problem Statement. Current quality analysis approaches ignore system commenting or are restricted to the comment ratio metric only. Hence, a major part of source code documentation is ignored during software quality assessment.

Contribution. Based on comment classification, we provide a semi-automatic approach for quantitative and qualitative evaluation of comment quality.

We present a semi-automatic approach for comment quality analysis and assessment. First, we perform comment categorization both for Java and C/C++ programs based on machine learning to differentiate between different comment types. Comment categorization enables a detailed quantitative analysis of a system's comment ratio and a qualitative analysis tailored to suit each single category. Comment categorization is the underlying basis of our comprehensive quality model. The model comprises quality attributes for each comment category based on four criteria: consistency throughout the project, completeness of system documentation, coherence with source code, and usefulness to the reader. To assess quality attributes, we provide metrics detecting quality defects in comments of specific categories. We evaluate the metrics' validity and relevance separately: Validity is evaluated with a survey among experienced software developers. The survey shows that the metrics can additionally give refactoring recommendations. A case study of five open source projects evaluates the relevance of our approach showing that comment classification provides better insights of the system documentation quality than the simple comment ratio metric and that our metrics reveal quality defects in practice.

II. Related Work

We group the related work into categories of general comment analysis, information retrieval techniques, comment evolution studies, and code recognition approaches.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant EvoCon, 01IS12034A. The responsibility for this article lies with the authors.

A. Comment Analysis

Khamis [9] et al. present the tool JavadocMiner to analyze the quality of Javadoc comments. With a set of simple heuristics, they aim to evaluate the quality of language used in comments and the consistency between source code and comments. They target the same research questions as we do: how to measure the comment quality. However, the authors did not evaluate whether metrics such as readability indices, noun and verb count heuristics, or abbreviation count heuristics can measure comment quality with meaningful results. Also the approach neither differentiates between different comment types nor detects any inconsistencies between code and comments beyond structural requirements of Javadoc.

Storey et al. [10] and Ying et al. [11] focus on the analysis of task comments. However, they do not provide automatic or semi-automatic assessments of task comment quality. In our work, we do not focus on task comments in particular but provide a general assessment of comment quality which also includes other comment categories.

Tan et al. [12] explore the feasibility and benefits of comment analysis to detect bugs in code. With their technique, they detected 12 bugs in the Linux kernel, two of them being confirmed by developers. They convincingly reveal the need for an automated comment analysis. However, the analysis is tailored to the specific topic of synchronization. In contrast, our approach analyzes comments independent of the context.

B. Information Retrieval Techniques

Lawrie et al. [13] use information retrieval techniques based on cosine similarity for vector space models to assess function quality under the guiding assumption that "if the code is high quality, then the comments give a good description of the code". Similar as in this work, we also investigate the similarity relation between source code and comments. However, we compare the relation between comment and method name and hence focus on a group of comments that has been ignored by this work. [14] and [15] also use information retrieval techniques to recover traceability links between code and documentation. However, comments do not play a major role in these approaches which rather focus on the general relation model between code and free text documentation.

C. Evolution of Code and Comments

Jiang and Hassan [16] study the evolution of comments over time to investigate the common claim that developers change code without updating its associated comments which is likely to cause bugs. However, the study reveals that developers update function comments regularly. Similarly, Fluri et al. [17] expect that code and comments are not necessarily updated at the same time and investigate how code and comments evolve. The study reveals that among all comment changes triggered by source code changes, about 97% are done in the same revision as the source code change. The authors also evaluate the comment ratio over time to give a trend analysis whether developers increase or decrease their effort on code commenting. However, as in [6], [7], commented out code or copyrights should be excluded in this metric because it does not provide any information gain for system understanding.

D. Source Code Recognition

Comment categorization (Section IV) includes source code recognition. To recognize code during email data cleaning, Tang et al. [18] use support-vector machines (SVMs), with a precision of 92.97%, and a recall of 72.17%. We apply similar machine learning techniques but achieve a higher recall while maintaining precision. In contrast, [19] provides faster lightweight techniques (regular expressions, pattern matching) to identify code within emails, leading to 94% precision and 85% recall. Precision and recall of this approach and our approach are approximately the same. However, we detect code snippets among comments and not among email data.

III. Approach

Our approach consists of four steps: We perform comment classification with machine learning to differentiate between different comment categories (IV). Based on the categories, we develop a comment quality model (Section V). For member and inline comments, we propose one metric each to assess parts of the model in practice (Section VI). We evaluate the metrics' validity with a survey (Section VII) and show the approach's relevance with a case study (Section VIII).

IV. Comment Classification

As different comment categories have the same underlying syntax, no parser or compiler can perform comment classification based on grammar rules. Hence, the problem requires a heuristic approach. We define seven different comment categories and employ machine learning techniques for automatic comment classification as there is no simple reliable classification criteria due to a large amount of features influencing the classification decision. For implementation, we use the existing machine learning library WEKA.[1]

A. Comment Categories

For the Java and C/C++ programming language, we differentiate between seven different types of comments:

- **Copyright comments** include information about the copyright or the license of the source code file. They are usually found at the beginning of each file.
- **Header comments** give an overview about the functionality of the class and provide information about, *e. g.*, the class author, the revision number, or the peer review status. In Java, headers are found after the imports but before the class declaration.
- **Member comments** describe the functionality of a method/field, being located either before or in the same line as the member definition. They provide information for the developer and for a project's API.
- **Inline comments** describe implementation decisions within a method body.

[1] http://www.cs.waikato.ac.nz/ml/weka/

- **Section comments** address several methods/fields together belonging to the same functional aspect. A fictitious example looks like

  `// ---- Getter and Setter Methods ---`

  and is followed by numerous getter and setter methods. .
- **Code comments** contain commented out code which is source code ignored by the compiler. Often code is temporarily commented out for debugging purposes or for potential later reuse.
- **Task comments** are a developer note containing a remaining todo, a note a about a bug that needs to be fixed, or a remark about an implementation hack.

B. Training Data

In general, a supervised machine learning algorithm learns a classification of an object based on a training data set. Training data is labeled with the classification to be learned and represented with features extracting relevant information. The success of machine learning depends on an appropriate feature representation which is not known a priori.

We created two separate training sets, one each for programs written in Java and C++. For Java, we randomly sampled files from twelve open source projects and manually tagged 830 comments with labels corresponding to the categories in IV-A. Table I shows the number of comments tagged per project and a project domain description and also presents the number of comments tagged in each category.[2] Analogous, we tagged about 500 comments in C++ code. More information about the C++ data is provided in [20].

C. Feature Extraction

Table II shows the features used for machine learning. As the optimal set of features is not known a priori, we experimented with a variety of features. However, the final decision tree revealed that only the subset of our initial set as presented in Table II is relevant for classification. The preliminary feature selection process can be found in [20].

As it is the most interesting example, we only explain the *code snippet* feature calculation which represents whether the comment contains commented out code. The *code snippet* feature is true if the ratio of lines of code to all lines of the comment is higher than a threshold. After preliminary experiments, we set the threshold to 0.1. A line is considered to contain code if one of the three characteristics is fulfilled:

- it matches the regular expression[3] for the Java method call pattern `[a-zA-Z]+\\.[a-zA-Z] +\\(.*\\)`
- it matches the regular expression of an `if` or `while` statement: `(if\\s*\\(.*)| (while\\s*\\(.*)`
- it ends with either `;` or { or contains ‘=, ==, `;`, or `void`

[2]One could argue that the Java classifier is trained predominantly to classify comments in jMol, as 389 comments were tagged in this project. However, we also trained the classifier without the jMol data. Considering the decrease in the size of the data set, the classifier still performed comparably well. We use comments from jMol to ensure enough data in categories that rarely occur in other projects such as tasks, commented out code, and section comments.

[3]We use the regular expression syntax as used in the java.util.regex package: http://docs.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html

TABLE I
JAVA TRAINING DATA SET GROUPED BY CATEGORIES AND PROJECTS

category	copyright	header	inline	member	section	code	task
Tags	61	48	297	271	65	73	15

Project	Version	Content	# Comments tagged
CSLessons	N.A.	simple code examples	39
EMF	2.7	modeling tool	24
Jung	2.0.1	framework for graph algorithms	32
ConQAT	2011.9	quality assessment toolkit for SE	89
jBoss	6.0.0.	application server	16
voTUM	0.7.5	visualization for compiler optimizations	109
			109
mylyn	3.6.0	monitoring tool	38
pdfsam	2.2.1	split and merge tool for pdfs	39
jMol	12.2	chemical visualization for 3D structures	389
jEdit	4.5	text editor	10
Eclipse	3.7	Compiler (eclipse.pde.core)	3
jabref	2.7.2	management tool for bibliographies	42

TABLE II
MACHINE LEARNING FEATURES FOR COMMENT CATEGORIZATION

Name	Type	Description
copyright	bool	true if comment contains “copyright” or “license”
braces	int	indicates how many braces are open at the position of the comment
decl. distance	int	measures the distance in lexical tokens to the next method/field declaration
frame	bool	true if comment contains a separator string multiple times (*e. g.*, “***”, “- - -”, “///”)
length	int	counts the number of words, separated by white spaces, in the comment
task tag	bool	true if comment is tagged with “task”, “fixme”, or “hack”
followed	bool	indicates whether the comment is directly followed by another comment
special characters	double	indicates the percentage of special characters in a comment (*e. g.*, “;”, “=”, “(”, “)”)
code snippet	bool	true if comment contains code snippets

The *code snippet* feature was designed for Java. However, we also use it for classifying C++ comments with reasonable success (see Section IV-D). Slight feature adaptations such as the C++ method call pattern probably improve C++ results.

D. Algorithms and Evaluation

Based on preliminary experiments [20], we chose the J48 decision tree algorithm [21] as it provided the most promising results. For evaluation, we use the standard five-fold cross validation method [22], calculating precision and recall. Table III shows that the algorithm performs comment classification successfully. In total, the decision tree achieves a weighted average precision and recall of 96%. For most categories, precision and recall are above 93%. For commented out code, precision drops below 90% because code is often misclassified as inline, member, or section comments and vice versa. Section

TABLE III
RESULTS FOR J48 ON JAVA (P: PRECISION, R: RECALL)

Class	P	R	Class	P	R
code	0.89	0.95	member	0.96	0.97
copyright	0.98	1	section	0.92	0.83
header	0.98	0.96	task	0.93	0.93
inline	0.98	0.98			

comments are also occasionally difficult to distinguish from member comments without deeper semantic knowledge.

With the same features, comment classification also works successfully in the C/C++ case: The J48 tree results in 95% precision and recall (weighted average). Although comment classification is the foundation for this work, it is not in the primary focus of this paper. Hence, we refer to more details about the C++ use case in [20].

V. QUALITY MODEL

Analyzing the quality of code comments requires a precise definition of comment quality. Our quality model resembles the quality models in maintenance [23]. Due to limited space, we only present a small part of the model providing context for the metrics proposed in Section VI.

The model is based on *entities*, *activities*, and *criteria*: Entities describe the concept whose quality is under evaluation *i. e.*, the comment categories. Activities represent the developers' intentions to comment code, *e. g.*, to better understand implementation details or to know bugs/hacks. We group activities in a hierarchical tree structure with an increasing level of detail from root to leaves as seen in Figure 1.

Criteria represent quality aspects for different entities and show the impact of an entity on a specific activity. This separation of concern reveals the general impact how commenting can support developers' activities. Our model comprises four criteria - *coherence*, *usefulness*, *completeness*, and *consistency*: We state the criteria more precisely by defining specific attributes for different entities (see Table IV). Each attribute creates either a positive impact (+) indicating that the entity (comment type) is meant to support the corresponding activity or a negative impacts (-) indicating a comment type hindering an activity. We observed and defined these impacts based on manual analysis and interpretation of the training data. In the following, we show how each attribute influences specific activities, grouped by the main four criteria.

Coherence covers aspects of how comment and code relate to each other, dealing with local aspects of a single comment. Member comments should be related to the method name as this is a strong indicator for an up-to-date comment and a meaningful method identifier. This supports calling public methods and understanding the system design. Further, developers expect member and inline comments to explain the non-obvious by providing information beyond the code to enhance understanding implementation and design details. Member comments in particular should provide more information than just repeating the method name.

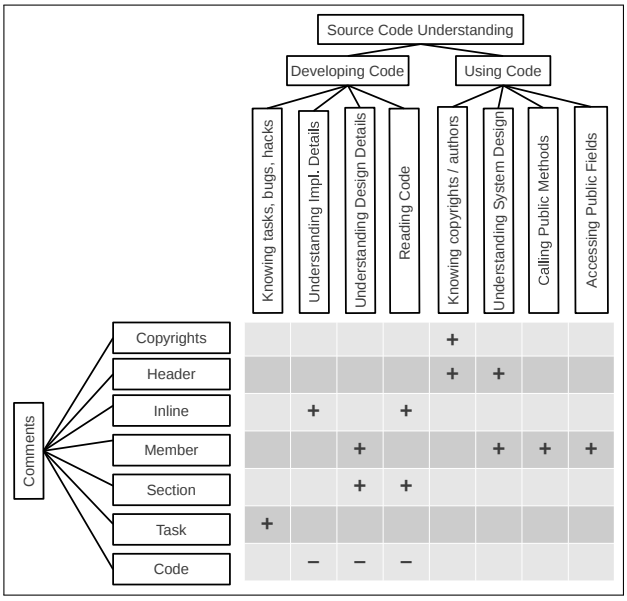

Fig. 1. Positive impacts (+) and negative impacts (-) of entities on activities

Usefulness describes properties that make a single comment contributing to system understanding. Comments should clarify the intent of code. In general, readers of the code should perceive the comment as helpful. If source code understanding was not harder with the comment being deleted, the comment would not be helpful. Clarifying, helpful comments make all activities about understanding and using code easier.

Completeness covers global aspects of system commenting by reinforcing comments at certain positions: placing a copyright in each file provides full information about copyrights, placing a header for each class documents the system design. Documenting every method and field with a member comment helps an API user to select public methods and fields.

Consistency attributes describe features that ought to be consistent through the system: Comments should be written in the same language (*e. g.*, English) for better code reading. Further, each file should be under the same copyright with a consistent format, promoting knowing copyrights and authors.

The impacts as described among the four quality criteria are visualized as a matrix in Figure 1. (In general, we consider code comments to have negative impacts as they do not provide any information.) The impact matrix reveals that only a subset of entities promote system understanding: header, member, inline, and section comments can possibly enhance activities such as calling methods/fields, understanding the system design, and implementation aspects (see Figure 1). Nevertheless, other comments (copyrights, tasks) are also an indispensable part of the documentation.

The quality model requires automatic assessment to be reinforced in practice. In this paper, we focus on the coherence category. Approaches for assessing consistency and usefulness can be found in [20].

TABLE IV
QUALITY CRITERIA

Coherence		Completeness	
Member	Related to method name	Copyright	for every file
All	Explaining non-obvious	Header	for every file
		Member	for every method

Consistency		Usefulness	
All	consistent language	All	clarifying
Copyright	consistent holder and format	All	helpful

VI. ASSESSMENT METRICS

In this section, we propose two different metrics for member and inline comments to assess coherence attributes.

In general, we preprocess comments before applying any metric: normalization removes commented out code as identified by the machine learning feature (Section IV) as well as Javadoc tags, comment delimiters ("/**", "//","*", "**/" etc.), line breaks, and special characters.

A. Coherence between Code and Comments

To measure the coherence between member comment and method name, we define a metric called *coherence coefficient* (c_coeff), which evaluates the attributes *explaining the non-obvious* and *related to method name* for member comments. Figure 2 shows an example where the comment explains the obvious and is hence unnecessary. The comments in Figure 3 and 4 are not related to the method name due to an non-informative comment (Fig. 3) or an non-informative identifier (Fig. 4). In the latter, the method should be better renamed to, *e. g.*, `calcEigenvalueDecomposition`. Our proposed metric will detect all three cases.

Metric. We extract words contained in the comment and compare it to the words contained in the method name. Words in the comment are assumed to be separated by white spaces, words in the method name are extracted using camel-casing. The comparison counts how many words from one set correspond to a similar word in the other set. Two words are similar iff their Levenshtein distance is smaller than 2. The c_coeff metric denotes the number of corresponding words divided by the total number of comment words. The comment in Figure 2, *e. g.*, has c_coeff$= 0.75$.

Based on preliminary experiments with manual evaluation, we set two thresholds and inspect member comments with c_coeff = 0 and c_coeff > 0.5.

Hypothesis 1. *Comments with c_coeff = 0 indicate that the coherence between method name and comment is not sufficient: They should have additional information to emphasize the relation to the method identifier or indicate a poor identifier.*

Hypothesis 2. *Comments with c_coeff* > 0.5 *are trivial as they do not contain additional information.*

We assume the middle group ($0 <$ c_coeff ≤ 0.5) to be a gray area with most comments containing additional information but some comments being trivial or unrelated. We expect comments with c_coeff $= 0$ to be a strong indicator for bad comment quality (insufficient information, Figure 3) or low code quality (poor method identifier, Figure 4) and comments with c_coeff > 0.5 to be trivial (Figure 2, c_coeff$= 0.75$).

```
/** removes all defined markers */
public void removeAllMarkers() { ... }
```

Fig. 2. Example of a trivial member comment

```
/** Thanks to benoit.heinrich on forum.java.sun.com
  * @param str
  */
private String HTMLEncode(String str){...}
```

Fig. 3. Example of a member comment that should provide more information

```
/** Check for symmetry, then construct the
  * eigenvalue decomposition
  * @param A square matrix
  */
public void calc(double[][] A) {
   for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {V[i][j]= A[i][j];}
   }
    tred2(); // Tridiagonalize.
    tql2(); // Diagonalize.
    ...
}
```

Fig. 4. Member comment where the method should be renamed

```
public void run(){
  if (server != null){
    final Thread shutdownThread = new Thread(){...}
    // Run
    shutdownThread.start();
    try { shutdownThread.join(); }
    catch (final InterruptedException ie) {...}
  }
}
```

Fig. 5. Example of a short inline comment that is unnecessary

B. Length of Comments

As the second metric, we use the length of inline comments as an indicator of their coherence to the following lines of code. This evaluates the quality attribute *explaining non-obvious* of inline comments. Intuitively, shorter inline comments contain less information than longer ones but the role of very short or long inline comments has not been investigated scientifically. Figure 5 shows a very short comment that is unnecessary as it explains the obvious. The comment in Figure 6 describes the functionality of the following `for`-loop. It would be better to extract the loop into a new method with the comment as method identifier. In contrast, the comment in Figure 7 contains non-obvious information that cannot be extracted from the following lines of code.

```
private CloneGroup canBeMerged(CloneGroup g1,
    CloneGroup g2) {
  CloneGroup merged = new CloneGroup();
  // merge
  for (int i = 0; i < g1.graphs.size(); i++) {
    ModifiableModelGraph mergedGraph = g1.graphs.get
        (i).copy();
    ModifiableModelGraph toMerge = g2.graphs.get(i);
    mergedGraph.merge(toMerge);
    merged.graphs.add(mergedGraph);
  }
  ...
  if (merged.isValid()) {return merged;}
  return null;
}
```

Fig. 6. Example of a short inline comment that indicates a method extraction

```
private void finishSaving(View view, String oldPath,
      String oldSymlinkPath, String path,
      boolean rename, boolean error) {
  if (!error && !path.equals(oldPath)){
    Buffer buffer = jEdit.getBuffer(path);
    if (rename){
      /* if we save a file with the same name as one
       * that's already open, we presume that we can
       * close the existing file, since the user
       * would have confirmed the overwrite in the
       * 'save as' dialog box anyway
       */
    if (buffer != null && !buffer.getPath().equals(
        oldPath)){
      buffer.setDirty(false);
      jEdit.closeBuffer(view,buffer);
    }
    ...
}
```

Fig. 7. Example of a long inline comment that contains global information

Metric. After normalization, the metric counts the number of words in a comment.

Based on preliminary experiments with manual evaluation, we set two thresholds and inspect inline comments with at most two and with at least 30 words.

Hypothesis 3. *Comments with at most two words should either be deleted or indicate that the following lines of code are better extracted into a new method with the comment content expressed in method name. They contain only information that can be extracted from the following lines of code.*

Hypothesis 4. *Programmers want to keep comments with at least 30 words. They contain information that can not be extracted from the following line(s) of code.*

Both hypotheses consist of two claims, each of which are to be evaluated independently. For comments with more than two, but less than 30 words we expect that the length of the comment can not be used as a reliable indicator for either the delete/keep decision or a decision about the information scope.

VII. Evaluation

We evaluated both metrics with a survey among developers. In the following, we describe the survey design before presenting the evaluation results. For each metric, we present the survey question, the sampling process of the comments presented in the survey, the results, and their implications.

A. Survey Design

The survey was designed in form of an online questionnaire, containing two independent evaluation tasks, which were both completed by 16 developers. The survey was sent to members of the Software and System Engineering Chair of Technische Universität München, TUM,[4] the Software Engineering group of the University of Bremen[5], students of TUM, employees of the software quality consulting company CQSE GmbH[6] and some of their clients. For each evaluation task, the participant was shown questions regarding his experience in programming and several example comments with their following or surrounding method, which were selected from the Java training data (Section IV-B). For evaluation (Figures 8 - 10), examples are numerated consecutively. In the survey, however, comments were displayed in random order.

Every participant had at least 5 years, 62% more than ten years of programming experience in general. 94% had at least 5 years of programming experience in Java. 69% use code from other projects, frameworks, or libraries frequently, 75% have worked on commercial, 88% on open source projects. Hence, although the number of participants is only 16, we assume that the participants are sufficiently experienced to provide an objective representative evaluation.

We visualize the answers with traffic light coloring: Red indicates low comment quality, green represents high comment quality. Yellow and orange symbolize moderate quality.

B. Evaluation of the Coherence Coefficient Metric

In order to evaluate the coherence coefficient, the survey asked the following question:

Survey Question 1. Please decide for each comment:

a) It would not make a difference if the comment was not there because the comment is trivial. It does not provide additional information which is not already contained in the method name.
b) The comment should have additional information because it is not obvious how comment and method name relate to each other.
or The method name could be more meaningful. The comment provides some useful information but a better method name could have been chosen.
c) The comment provides additional information, which is not contained in the method name, and the method name is meaningful.

[4] http://www4.in.tum.de/
[5] http://www.informatik.uni-bremen.de/st/index.php
[6] www.cqse.eu

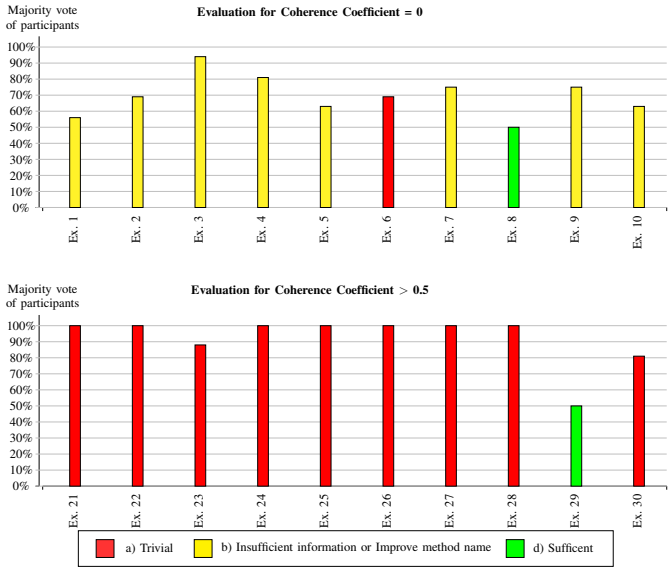

Fig. 8. Survey results for comments with c_coeff $= 0$ and c_coeff > 0.5

Additionally, the question provides the answer possibility "other" where the participant is able to enter a comment.

Based on this survey question, Hypothesis 1 is fulfilled if developers vote for answer *b)* when evaluating comments with c_coeff $= 0$. Developers voting for *a)* when evaluating comments with c_coeff > 0.5 support Hypothesis 2.

Sampling. To evaluate the hypotheses, we sampled member comments as identified by the machine learning classifier according to their coherence coefficient: we grouped them into three categories based on the two thresholds 0 and 0.5. From each group we randomly sampled ten comments. In the questionnaire, we display all 30 comments in random order to not influence the opinion of the participants.

Results. Diagrams 8 shows the results of the survey, indicating the majority vote of the participants. For eight out of ten comments with c_coeff $= 0$, the participants voted with absolute majority for an insufficient relation between comment and method name (*b*). In five cases, the participants indicated that the comment is missing relevant information. In three cases, they preferred to rename the method. This result strongly supports Hypothesis 1. In nine out of ten examples with c_coeff> 0.5 (Figure 8), the voters agreed on the comments being trivial without information gain for system commenting (*a*). In all nine cases, the agreement was with above 80% very strong. This strongly confirms Hypothesis 2.

Implications. The c_coeff metric is a useful metric to detect trivial comments fully automatically, tolerating a correctness of 90%. For comments with an insufficient coherence, the developer still needs to decide manually whether the method should be renamed (code refactoring) or information should be added to the comment (documentation improvement), which results in a semi-automatic analysis.

C. Evaluation of the Length Indicator

In order to evaluate the length indicator, we asked the developers the following two survey questions:

Survey Question 2. If you work on improving the quality of system commenting and you have the following choices, which one do you pick?

a) Delete the comment because it is redundant and unnecessary. Without the comment the source code would not be harder to understand.
b) Remove the comment by extracting a method. The lines of code following the comment can be extracted into a new method by using the content of the comment as a method name. After this refactoring, the comment would not exist anymore but be expressed in the method name.
c) Keep the comment. The comment helps to understand the code and the system.

Survey Question 3. Please decide:

d) The comment contains some information which can not be extracted from the following line(s) of code (global information, explanations beyond the scope of the current method, general assumptions, design decisions, information describing the system's behavior, information not obvious to express in code, potential risks or failures etc.)
e) The comment contains only information which can be extracted from the following line(s) of code.

We refer to the information described in *d* as global, the one in *e* as local information.

For comments with at most two words, developers voting for answers *a* and *b* validate part one of Hypothesis 3, *e* suppots part two. For comments with at least 30 words, votes for *c* and *d* support part one and two of Hypothesis 4.

Sampling. We sampled inline comments from the Java training data set, excluding jMol. Beforehand, manual inspection revealed that inline comments from jMol have significantly less quality than inline comments in other projects, including but not limited to being out-of-date, at the wrong place, or just not understandable. From all other projects, we selected randomly five inline comments. Out of this pool, we sampled ten inline comments with at most two words, ten with ≥ 30 words, and ten with > 2 and < 30 words. Sampling from the preselected pool of comments ensures a uniform distribution over the quality of inline comments among different projects.[7]

Results. Figure 9 represents the results of Question 2. They visualize the vote distribution over all three answers, unifying answer *a* and *b* because both represent a comment deletion. We take the majority vote criterion to determine the final

[7]We could have sampled ten comments per category from all inline comments of all projects. However, as some projects contain significantly more inline comments than others, the survey might have been biased towards the comment quality of a few individual projects instead of being representative for all projects.

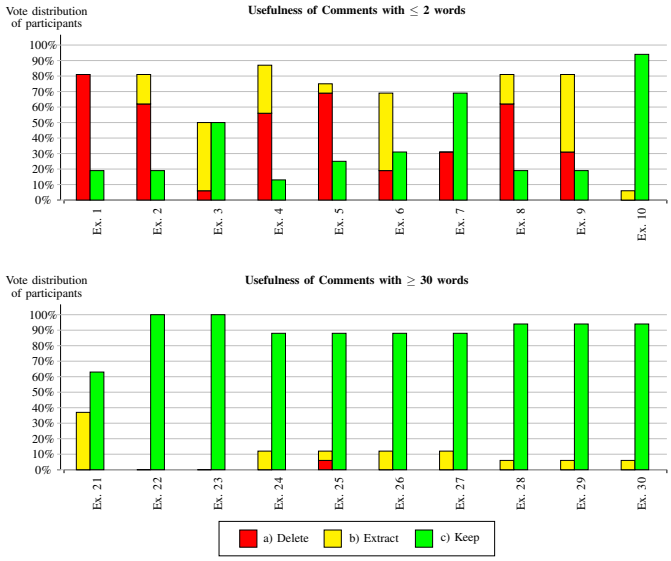

Fig. 9. Survey participants' decisions for short and long inline comments

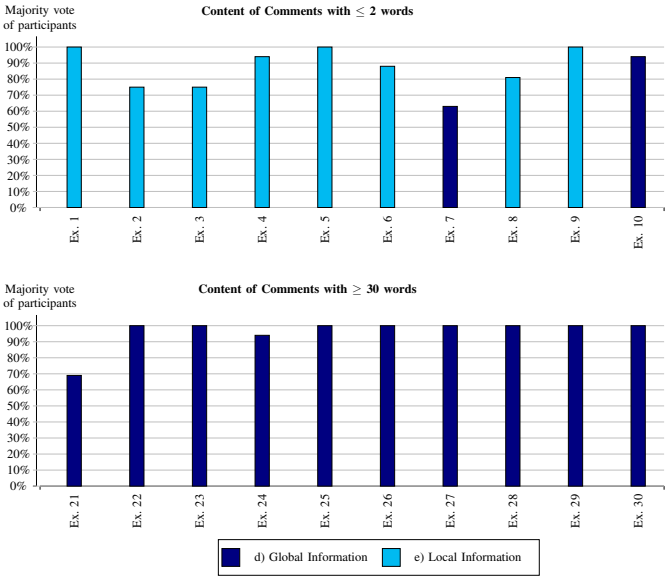

Fig. 10. Survey results about information content in short and long inline comments

participants' answer per example. For comments with at most two words, in five cases, the majority of participants voted for deleting the comment. In two cases, 50% of the participants wanted to extract a new method and thereby remove the comment. Summing up, in 70% of all examples, the participants removed it directly or by method extraction. This supports the first part of Hypothesis 3. An example (Ex. 10) that did not concur with the hypothesis is shown in Figure 11. For comments with at least 30 words, Figure 9 shows that in all ten cases developers wanted to keep the comment, with a very high agreement among each other of at least 88% in nine out of ten cases. This strongly validates part one of Hypothesis 4. As expected, the survey did not reveal a clear programmers' preference how to handle comments with more than two but less than 30 words.

Figure 10 reveals that eight out of ten comments with at most two words contain only local information with an agreement of at least 75%. In contrast, comments containing at least 30 words contain global information in ten out of ten examples, with an agreement among voters of at least 92% in nine cases (Figure 10). Both facts support part two of Hypothesis 3 and 4. Comments between two and 30 words contain both local and global information.

Implications. The length indicator can be used fully-automatically to detect comments with at most two words strongly indicating redundant local information that should be deleted. This metric can be used to suggest developers where lines of code should be extracted into a new method and hence give refactoring recommendations.

In contrast, comments with more than 30 words contain significant global information which promotes better understanding of the system. It is inappropriate to conclude that inline comments should contain at least thirty words. In general, they should not. However, for semi-automatic assessment,

```
JARClassLoader(boolean delegateFirst){
  this.delegateFirst = delegateFirst;
  // for debugging
  id = INDEX++;  live++;
}
```

Fig. 11. Inline comment with two words that developers wanted to keep.

their absence can potentially indicate that global aspects of the system are not commented. It remains to verify if global aspects are documented elsewhere (*e. g.*, in header comments or external system documentation). Vice versa, too many very long inline comments can signal that too much global information is documented in comments. This can indicate that either architecture descriptions or framework documentation of the system are missing, domain-specific knowledge is not documented externally, or the system design is too general.

VIII. Case Study

Whereas the previous survey evaluated the correctness of the proposed assessment metrics, we additionally conduct a case study to evaluate the relevance of the entire model, showing its use case in practice. The following research questions guided the design of the case study:

RQ 1: How many comments do not enhance system understanding? We investigate if applying the comment classification provides different insights than measuring the comment ratio as in previous literature.

RQ 2: Do the proposed metrics reveal quality defects in practice? We calculate the coherence coefficient for member comments and the length indicator for inline comments to show quality defects addressed by these two metrics frequently occur in practice.

TABLE V
RESULTS OF CASE STUDY

		jMol	ConQAT	jEdit	voTUM	JUNG
	CR	25%	38%	36%	29%	33%
Classification	Copyright	28%	49%	31%	39%	18%
	Header	5%	16%	10%	9%	20%
	Member	28%	28%	33%	38%	49%
	Inline	21%	4%	10%	8%	6%
	Section	6%	2%	13%	2%	2%
	Task	0%	0%	0%	0%	0%
	Code	11%	0%	3%	3%	4%
	CR*	15%	19%	24%	17%	25%
Coherence	member trivial	42 (2%)	1774 (9%)	168 (5%)	15 (1%)	30 (1%)
	member unrelated	690 (39%)	1679 (9%)	897 (26%)	251 (19%)	504 (23%)
	inline short	2945 (44%)	785 (24%)	535 (21%)	215 (29%)	69 (9%)
	inline long	296 (5%)	80 (2%)	88 (3%)	17 (2%)	10 (1%)

The purpose of this case study is not to provide an overall ranking of the comment quality between the five systems. Instead, we show that our approach is a very useful tool to provide a first semi-automated step for quality analysis.

A. Case Study Design

We chose five open source projects (jMol, ConQAT, jEdit, voTUM, JUNG) that were also used in Section IV-B.[8] For RQ 1, we measure the comment ratio (*CR*) as percentage of characters in source code which belong to comments. The classification shows the percentage of comment characters belonging to different comment categories. *CR** denotes the character percentage that potentially contributes to system understanding (header, member, inline, and section). For RQ 2, we present results from the c_coeff metric and the length indicator. We denote the percentage of all member comments (measured in number of comments), which are trivial or not related to the method name and the percentage of all inline comments which are too long ($\geq$ 30 words) or too short ($\leq$ 2 words). We also show the absolute finding numbers.

B. Results

RQ 1. To answer Research Question 1, we first calculate the simple *CR* metric, analyze the results of comment classification and then compare the *CR** with the *CR* metric.

Calculating the simple *CR* metric reveals (Table V) that ConQAT has the highest comment ratio (38%), followed by jEdit (36%) and JUNG (33%). The classification provides more insights about the types of comments that do not promote system understanding (copyrights, tasks, and commented out code): Among all projects, between 18% (JUNG) and 49% (ConQAT) of the comments are copyrights. Although ConQAT has the highest overall comment ratio, at least half of the comments do not enhance documentation quality. The percentage of task comments is insignificantly small for all projects.[9] Furthermore, between 0% of the comments (ConQAT) and 11% (jMol) are commented out code.

The remaining comment categories (header, member, inline, section) potentially contribute to documentation quality. Among all projects, jMol has the highest percentage of inline comments (21%) and section comments (6%). Manual inspection suggests that jMol depends on these comments as the average method length and file size are too large. The analysis also reveals that jMol has deficits in documenting the system design as only 5% are header comments. (A completeness analysis confirms this as only 37% of all files are documented with a header comment.) In contrast, ConQAT is barely documented with inline and section comments, but mostly in header comments (16%) and member comments (28%). JEdit has the highest amount of section comments (13%) because a grouping style is used that spans groups of method with a //{{{ and //}}} comment even if the group contains a single method. VoTUM and JUNG have the highest percentage of member comments (38% for voTUM, 49% for JUNG) due to their framework/library character.

With this classification, we calculate the CR* metric which excludes categories that do not promote system understanding. The CR* metric shows that at most 25% of all source code characters in JUNG can enhance the system documentation quality, followed by 24% for jEdit, and 20% for ConQAT. Whereas the simple comment ratio metric ranked ConQAT as the best documented system, JUNG has the highest percentage of comments potentially contributing to system understanding. The CR* metric shows that comment classification is necessary to differentiate between different comment types to get more quantitative insights about a system commenting.

We do not claim that the CR* metric should be used in the future as the one and only metric to measure comment quality. This would not be sufficient but the CR* metric constitutes one aspect of a thorough comment analysis, providing better insights than the simple comment ratio.

RQ 2. Applying the c_coeff-metric and the length indicator reveals defects in system commenting and gives refactoring recommendations. For example, ConQAT has the highest relative number of trivial member comments (9%) which results in an absolute number of 1774 comments with c_coeff>0.5 because project settings produce warnings when member comments are missing: To avoid a compiler warning, developers prefer to write a quick, but trivial member comment. These comments should be reinspected to add missing information.

All projects have a percentage of unrelated member comments between 9% (ConQAT) and 26% (jEdit). Developers should check these findings in terms of identifier quality of the method and information content of the comment.

[8] With ConQAT, we use a tool as study object developed by our own research group. However, the first author of this paper performing the study was not an active maintainer of ConQAT and hence objective enough to provide a fair evaluation.

[9] For all projects the task comment percentage was below 0.5% and, hence, rounded to 0% in Table V.

An analysis of inline comments in jMol shows that 2945 have at most two words (44%), indicating a very low quality of inline documentation. These comments should be reinspected and either be deleted or used for the name of a newly extracted method. 296 inline comments have at least 30 words (5%). Manual inspection reveals that developers should document the system better in an external architecture description as these comments contain useful information but should mostly not be placed in the source code.

Using the c_coeff-metric and the length indicator results in a significant number of findings showing that the addressed quality defects frequently occur in practice. These findings are a first suggestion to developers how to improve comment quality and also how to refactor the source code.

IX. Application

This work provides a first approach towards a thorough analysis of code comments. In this section, we evaluate the strengths and weaknesses of our analysis. Our analysis provides quantitative and qualitative assessment of comment quality: Compared to the previously used comment ratio, our comment classification provides better quantitative information about system commenting. Compared to related work, we suggest two metrics to detect quality defects in comments and to give refactoring recommendations whose correctness and relevance were evaluated with a survey and a case study.

Comment classification and the suggested metrics are applicable for a one-time quality audit or for continuous quality control to give a trend analysis about system commenting. The c_coeff metric and the length indicator can be used to display warnings in the IDE during development to call the developer's attention to insufficient comment or code quality.

The suggested metrics are semi-automatic and require manual inspection which is, however, already often included in the use-case of quality control. The approach as suggested in our paper cannot be used to rank the comment quality of several projects and also does not provide a complete comment quality assessment. However, this work is the foundation for a thorough quality analysis by providing comment classification and a quality model. Similar as for code quality, our metrics only reveal defects but cannot detect high quality directly.

X. Conclusion and Future Work

This work presented a first detailed approach for the analysis and assessment of code comments. A machine-learning approach for comment classification provides the foundation for a model of comment quality. The model describes detailed quality attributes in terms of coherence, consistency, completeness, and usefulness. With the coherence coefficient and the length indicator, we provided two metrics to assess quality attributes of the model and evaluated them with a survey among experienced developers. The case study demonstrated that our approach can provide a much more detailed analysis compared to existing approaches: Comment classification provides better quantitative insights about the system documentation as the simple comment ratio metric. For a qualitative analysis, the suggested metrics reveal quality defects in code commenting that frequently occur in practice. Furthermore, the metrics can also be used to give refactoring recommendations. The length indicator, *e. g.*, suggests to extract methods when inline comments with at most two words are used. The c_coeff metric detects both comments with insufficient information and methods with a low-quality method identifier.

This work constitutes the foundation for future comment quality assessment. More work is required to find additional comment quality metrics and to determine how many more aspects of comment quality can be assessed fully- or semi-automatic and how many remain only manually assessable.

References

[1] T. Tenny, "Program Readability: Procedures Versus Comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, 1988.

[2] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," ser. ICSE '81, 1981.

[3] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A Study of the Documentation Essential to Software Maintenance," ser. SIGDOC '05, 2005.

[4] C. S. Hartzman and C. F. Austin, "Maintenance productivity: Observations based on an experience in a large system environment," ser. CASCON '93, 1993.

[5] B. P. Lientz, "Issues in Software Maintenance," *ACM Computing Surveys*, vol. 15, no. 3, 1983.

[6] M. J. B. García and J. C. G. Alvarez, "Maintainability as a Key Factor in Maintenance Productivity: A Case Study," ser. ICSM '96, 1996.

[7] P. Oman and J. Hagemeister, "Metrics for Assessing a Software System's Maintainability," ser. ICSM '92, 1992.

[8] I. S. Microsystems, *Code Conventions for the Java Programming Language*, 1997. [Online]. Available: http://www.oracle.com/technetwork/java/codeconv-138413.html

[9] N. Khamis, R. Witte, and J. Rilling, "Automatic Quality Assessment of Source Code Comments: the JavadocMiner," ser. NLDB '10, 2010.

[10] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers," ser. ICSE '08, 2008.

[11] A. T. T. Ying, J. L. Wright, and S. Abrams, "Source code that talks: an exploration of Eclipse task comments and their implication to repository mining," ser. MSR '05, 2005.

[12] L. Tan, D. Yuan, and Y. Zhou, "HotComments: How to Make Program Comments More Useful?" ser. HOTOS '07, 2007.

[13] D. J. Lawrie, H. Feild, and D. Binkley, "Leveraged Quality Assessment using Information Retrieval Techniques," ser. ICPC '06, 2006.

[14] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing," ser. ICSE '03, 2003.

[15] G. Antoniol, G. Canfora, A. de Lucia, and G. Casazza, "Information Retrieval Models for Recovering Traceability Links between Code and Documentation," ser. ICSM '00, 2000.

[16] Z. M. Jiang and A. E. Hassan, "Examining the Evolution of Code Comments in PostgreSQL," ser. MSR '06, 2006.

[17] B. Fluri, M. Wursch, and H. C. Gall, "Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes," ser. WCRE '07, 2007.

[18] J. Tang, H. Li, Y. Cao, and Z. Tang, "Email data cleaning," ser. KDD '05, 2005.

[19] A. Bacchelli, M. D'Ambros, and M. Lanza, "Extracting Source Code from E-Mails," ser. ICPC '10, 2010.

[20] D. Steidl, "Quality analysis and assessment of code comments," Master's Thesis in Computer Science, 2012, available online at http://www4.in.tum.de/~hummelb/theses/2012_steidl.pdf; visited on January 16th 2012.

[21] R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[22] R. Kohavi, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," ser. IJCAI '95, 1995.

[23] F. Deissenböck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard, "An Activity-Based Quality Model for Maintainability," ser. ICSM '07.