International Journal of Software Engineering
and Knowledge Engineering
Vol. 26, No. 6 (2016) 981–1000
© World Scientific Publishing Company
DOI: 10.1142/S0218194016500339

Code Comment Quality Analysis and Improvement Recommendation: An Automated Approach

Xiaobing Sun[*,§,¶], Qiang Geng[*], David Lo[†], Yucong Duan[‡],
Xiangyue Liu[*] and Bin Li[*]

[*]*School of Information Engineering*
Yangzhou University, Yangzhou, China

[†]*School of Information Systems*
Singapore Management University, Singapore

[‡]*Department of Computer Science, Hainan University*
Haikou, China

[§]*State Key Laboratory for Novel Software Technology*
Nanjing University, Nanjing, China
[¶]*sundomore@163.com*

Received 4 May 2015
Revised 25 June 2015
Accepted 7 September 2015

Program comprehension is one of the first and most frequently performed activities during software maintenance and evolution. In a program, there are not only source code, but also comments. Comments in a program is one of the main sources of information for program comprehension. If a program has good comments, it will be easier for developers to understand it. Unfortunately, for many software systems, due to developers' poor coding style or hectic work schedule, it is often the case that a number of methods and classes are not written with good comments. This can make it difficult for developers to understand the methods and classes, when they are performing future software maintenance tasks. To deal with this problem, in this paper we propose an approach which assesses the quality of a code comment and generates suggestions to improve comment quality. A user study is conducted to assess the effectiveness of our approach and the results show that our comment quality assessments are similar to the assessments made by our user study participants, the suggestions provided by our approach are useful to improve comment quality, and our approach can improve the accuracy of the previous comment quality analysis approaches.

Keywords: Program comprehension; code comment quality analysis; user study.

1. Introduction

During software maintenance and evolution, program comprehension is one of the most frequently performed activities [1]. Developers working on software maintenance tasks spend around 60% of their time comprehending a system [2]. The quality of a program affects the amount of effort, that needs to be invested to comprehend the program [3, 1, 4]. A program can be divided into source code (which consists of machine-translatable instructions) and comments (which include human-readable notes and other kinds of annotations) [5]. Thus, code comment quality influences program comprehension cost.

Comments are usually added with the purpose of making a piece of source code (e.g. a class or a method) easier to understand. If a program has good comments, it will be easier for developers to understand it. Unfortunately, not all developers write high quality comments. To improve code comment quality, several approaches have been proposed to assess code comment quality [6–8]. The latest approach by Steidl *et al.* assesses the quality of method comments, i.e. comments that describe the functionality of a method are typically located, just before or on the same line as a method definition [8]. Their approach computes the ratio between the number of similar words[a], that appear in both method comment and method name, and the total number of words in the method comment. If this ratio (aka., c_coeff) is zero or larger than 0.5, a method comment is deemed to be of poor quality. In a program, comment quality includes both existing comment quality and the situation that there is no comments but this code does need some necessary comments (for example, *authorship information*), which also affects comment quality [9].

In this paper, we extend the work of Steidl *et al.* so that a more accurate and comprehensive comment assessment and improvement recommendation can be made. Our approach analyzes not only method comments but also header/class comments (i.e. comments at the beginning of a class file). For these two types of comments, our approach analyzes the program elements with comments as well as those without any comments. Specifically, for a header comment, to assess its quality, we analyze whether it contains authorship information and whether the words in the comment correlate with the words that appear in the name of the corresponding class. For a method comment, to assess its quality, we analyze whether the method requires comments, and whether words in the comment correlate with words that appear in the name of the method. Some methods are simple methods (e.g. getName()) and for such methods, no comment is needed. Aside from assessing the quality of each individual header and method comment, we also compute an aggregate measure that assesses the quality of all of the code comments in a program file.

To assess the effectiveness of our approach, we perform a user study with 20 participants. We ask our participants to evaluate a number of code comments, that appear in the jdk8.0 and jEdit. We first, investigate, whether some hypotheses that underlie our approach are well supported by our user study participants or not. Second, we compare our participants manual assessment with automated assessment by our approach. Third, we use our approach to guide a developer to improve code

[a] Two words are deemed similar, if they have a Levenshtein distance of less than 2.

comments and evaluate if the guidance is useful. Finally, we compare code quality assessment produced by our approach with other code quality assessment approaches. The user study results show that our approach can reasonably assess the quality of code comments and produce good improvement recommendations. The effectiveness of our approach is also better than the state-of-art approaches.

The rest of our paper is organized as follows: Section 2 introduces our approach. The design of our user study is presented in Sec. 3 and the results of the user study are described in Sec. 4. In Sec. 5, we discuss some related work. Finally, we conclude our paper and discuss future work in Sec. 6.

2. Approach

In this section, we provide details of our approach which assesses the quality of a header comment, a method comment, and comments that appear in a program file. Header comments are typically used to provide an overview about the functionality of the class and some development information, e.g. the class authorship. These header comments are usually located after the import statements but before the class declaration[b] [10]. For member method comments, they are used to describe the functionality of a method. They are located either, before or in the same line as the method definition [10].

Figure 1 shows the process that is followed by our approach. First, we preprocess the source code, and divide the source code into header comments and member method comments. Then, we analyze the header and member method comments quality, and provide some suggestions for low quality comments. Finally, we analyze the quality of comments that appear in a file based on the quality of the header and method comments that appear in the file.

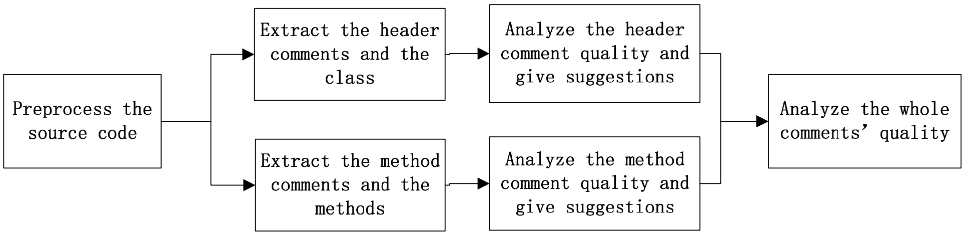

Preprocess the source code
Extract the header comments and the class
Extract the method comments and the methods
Analyze the header comment quality and give suggestions
Analyze the method comment quality and give suggestions
Analyze the whole comments' quality

Fig. 1. Steps in our proposed approach.

2.1. *Header comment analysis*

Header comments provide an overview about the functionality of the class, the authorship information, the revision number, etc. To analyze the header comments,

[b]In this paper, we only analyze Java programs.

we need some preprocessing operations on the class identifier and the header comments. We extract words contained in the class name using camel-casing. And words in the header comments are assumed to be separated by white spaces.

We consider the *authorship* information in a header comment to be important, since this information can be used during software evolution to identify author(s), that can help in various maintenance tasks, cf. [11]. If a header comment does not have this information, our approach will generate a suggestion, "*Please add authorship information to the header comment in this file.*". Aside from authorship information, a header comment should also provide sufficient description of the class. When the header comment does not provide any information, our approach will generate a suggestion, "*Please add information to the header comment to describe the class.*". If the header comments provide this information, we will further analyze the correlation between the header comment and the class name. For simplicity, the correlation is computed as the ratio of the number of class words that appear in header comments after preprocessing. Then, we compare it with a threshold value θ_{class}. If it is smaller than θ_{class}, our approach will give a suggestion, "*Please add some other relevant information about the class to the header comment.*".

2.2. *Method comment analysis*

Method comments provide information about the functionality or usage of the methods. We analyze method comments considering two aspects. First, we analyze methods with some comments to assess the comment quality. Second, we also analyze the methods without any comments to assess, whether they need some comments. In addition, some simple methods do not require any comment.

For a method with a comment, we compute the correlation between the method's name and the comment, and compare the correlation with a threshold value θ_{method}. We compute the correlation between a method's name, and its comment in the same way as, we compute the correlation between a class's name and comment. If the correlation is bad (its value is below the threshold), we generate a suggestion: "*Please add some more information to this method's comment to describe the method.*".

For a method with no comment, we count the number of lines of code in it and compare this number with a threshold θ_{line}. If the number of lines of code is greater than θ_{line}, we will generate a suggestion: "*Please add some comments to describe this method because there are more than θ_{line} lines of code in this method.*". The rationale of this recommendation is that long methods are typically harder to understand than short methods and thus they should be commented to make them easier to be understood. In addition, if the number of lines of code is smaller than the threshold value θ_{line}, we will count the number of method invocations inside the method and compare this number with the threshold value θ_{call}. If the number of method invocations is greater than θ_{call}, we will generate a suggestion, "*Please add some comments to describe this method since the method seems to be a complex one with more than θ_{call} method invocations.*".

2.3. *Quality assessment for comments in a file*

To quantitatively evaluate the quality of all the comments that appear in a file (SCF), we compute what we refer to as the *suggestion-rate* metric defined in the following equation:

$$\mathit{Suggestion\text{-}rate}(\mathrm{SCF}) = \frac{1}{3}\left(\frac{\sum_{i=1}^{n} SFH_i}{2} + \frac{\sum_{i=1}^{n} SFM_i}{\sum_{i=1}^{n} MC_i} + \frac{\sum_{i=1}^{n} SFNM_i}{\sum_{i=1}^{n} NMC_i}\right).$$

In the equation above, $\sum_{i=1}^{n} SFH_i$ represents the number of header comment suggestions that our approach gives. For the header comments, we consider two cases for which suggestions are made: the absence of the authorship information and inadequate description of the class. Thus, in the equation, we divide $\sum_{i=1}^{n} SFH_i$ by two. Next, $\sum_{i=1}^{n} SFM_i$ represents the number of suggestions that our approach gives to methods with existing comments, while $\sum_{i=1}^{n} MCi$ represents the number of methods that have comments. Thus, $\frac{\sum_{i=1}^{n} SFM_i}{\sum_{i=1}^{n} MC_i}$ represents the suggestion-rate of our approach for methods with comments. Finally, in the last part of the equation above, $\sum_{i=1}^{n} SFNM_i$ represents the number of suggestions that our approach gives to methods with no comments, while $\sum_{i=1}^{n} NMCi$ is the number of methods that do not have any comments. Thus, $\frac{\sum_{i=1}^{n} SFNM_i}{\sum_{i=1}^{n} NMC_i}$ represents the suggestion-rate of our approach for methods with no comments.

There are some special cases; first, if $\sum_{i=1}^{n} SFM_i = 0$, the suggestion-rate is computed by the following formula:

$$\mathit{Suggestion\text{-}rate}(\mathrm{SCF}) = \frac{1}{2}\left(\frac{\sum_{i=1}^{n} SFH_i}{2} + \frac{\sum_{i=1}^{n} SFNM_i}{\sum_{i=1}^{n} NMC_i}\right).$$

Also, if $\sum_{i=1}^{n} SFNMi = 0$, the suggestion-rate is computed by the following formula:

$$\mathit{Suggestion\text{-}rate}(\mathrm{SCF}) = \frac{1}{2}\left(\frac{\sum_{i=1}^{n} SFH_i}{2} + \frac{\sum_{i=1}^{n} SFM_i}{\sum_{i=1}^{n} MC_i}\right).$$

After computing the *Suggestion-rate*(SCF) for a source code file, we compare the rate with a threshold value θ_{file}. If *Suggestion-rate*(SCF) is larger than θ_{file}, the comment quality of this file is considered bad. Otherwise, its comment quality is considered to be good.

2.4. *Comment quality metrics and hypotheses*

In our approach, we compute some metrics, for example, existence of authorship information, correlation between header comments and the class, etc., to assess comment quality. In this section, we describe each individual metric and how each of them is computed. We also present a set of hypotheses that are derived based on these metrics.

2.4.1. *Existence of authorship information*

Authorship in a file represents the owner of (or developer in-charge of) this code file. We define a metric called *existence-authorship* which evaluates whether a header comment has authorship information or not. Figures 2(a) and 2(b) show the examples of header comments with and without authorship information, respectively.

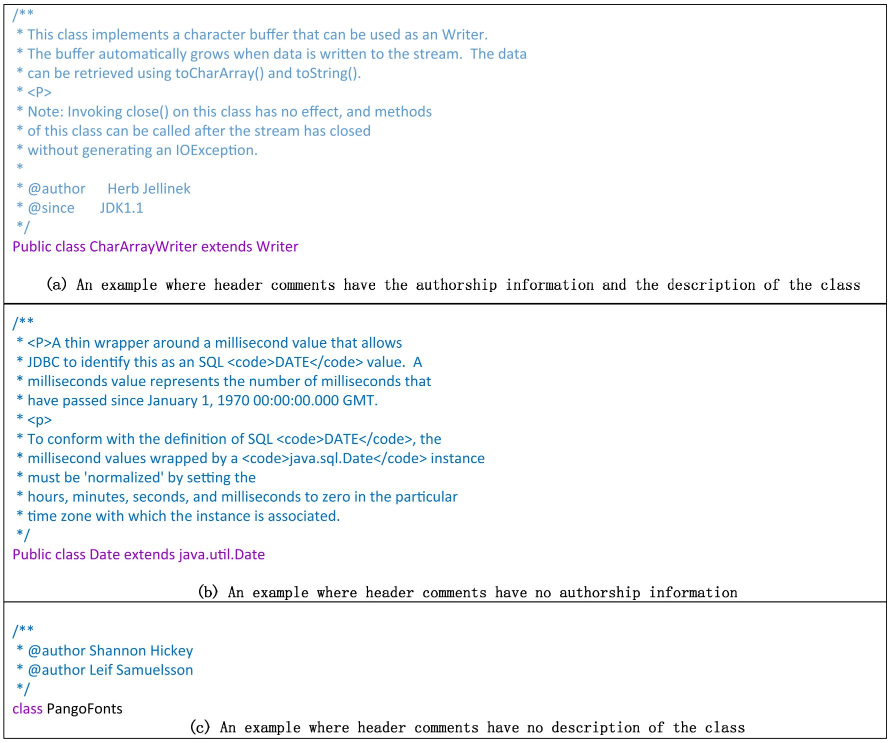
Metric Computation. We first extract a code file's header comment. If the header comment has the word "*@author*" and there are some words (author's name) after word "@author", *existence-authorship* = 1. Otherwise, *existence-authorship* = 0. Thus, the comment in Fig. 2(a) has *existence-authorship* = 1, and the comment in Fig. 2(b) has *existence-authorship* = 0.

Hypothesis 1. Header comments with *existence-authorship* = 0 are considered to be bad. Authorship information should be added.

Hypothesis 2. Header comments with *existence-authorship* = 1 are good.

2.4.2. *Correlation between header comments and class name*

To measure the correlation between header comments and class, we define a metric called *correlation-headerclass*, which evaluates the relevance of the class name to



```
/**
 * This class implements a character buffer that can be used as an Writer.
 * The buffer automatically grows when data is written to the stream.  The data
 * can be retrieved using toCharArray() and toString().
 * <P>
 * Note: Invoking close() on this class has no effect, and methods
 * of this class can be called after the stream has closed
 * without generating an IOException.
 *
 * @author     Herb Jellinek
 * @since      JDK1.1
 */
Public class CharArrayWriter extends Writer
```

(a) An example where header comments have the authorship information and the description of the class

```
/**
 * <P>A thin wrapper around a millisecond value that allows
 * JDBC to identify this as an SQL <code>DATE</code> value.  A
 * milliseconds value represents the number of milliseconds that
 * have passed since January 1, 1970 00:00:00.000 GMT.
 * <p>
 * To conform with the definition of SQL <code>DATE</code>, the
 * millisecond values wrapped by a <code>java.sql.Date</code> instance
 * must be 'normalized' by setting the
 * hours, minutes, seconds, and milliseconds to zero in the particular
 * time zone with which the instance is associated.
 */
Public class Date extends java.util.Date
```

(b) An example where header comments have no authorship information

```
/**
 * @author Shannon Hickey
 * @author Leif Samuelsson
 */
class PangoFonts
```

(c) An example where header comments have no description of the class

Fig. 2. Examples of header comments.

header comment. The comments in Fig. 2(a), show an example where the correlation between header comments and the class is high. The comments in Fig. 2(c), show an example, where header comments should be added with other description about the main class.

Metric Computation. We extract words contained in the class name and compare it with the words contained in the header comments. Then, we count how many words in the class name are also included in the header comments. The *correlation-headerclass* denotes the percentage of the number of class' words contained in the header comments divided by the total of class's words. Finally, we compare the result with the threshold θ_{class}. For example, header comments in Fig. 2(a) have *correlation-headerclass* $= 1$, and *correlation-headerclass* $= 0$ in Fig. 2(c).

Based on some experiments with manual evaluation, we set the threshold θ_{class} as 0.5. So, we have,

Hypothesis 3. Header comments with *correlation-headerclass* < 0.5 indicate that the correlation between header comments and class is bad. They should add other description about the class in the header comments.

Hypothesis 4. Header comments with *correlation-headerclass* ≥ 0.5 is assumed to be good in that the correlation between header comments and class is high.

2.4.3. *Correlation between method comments and method name*

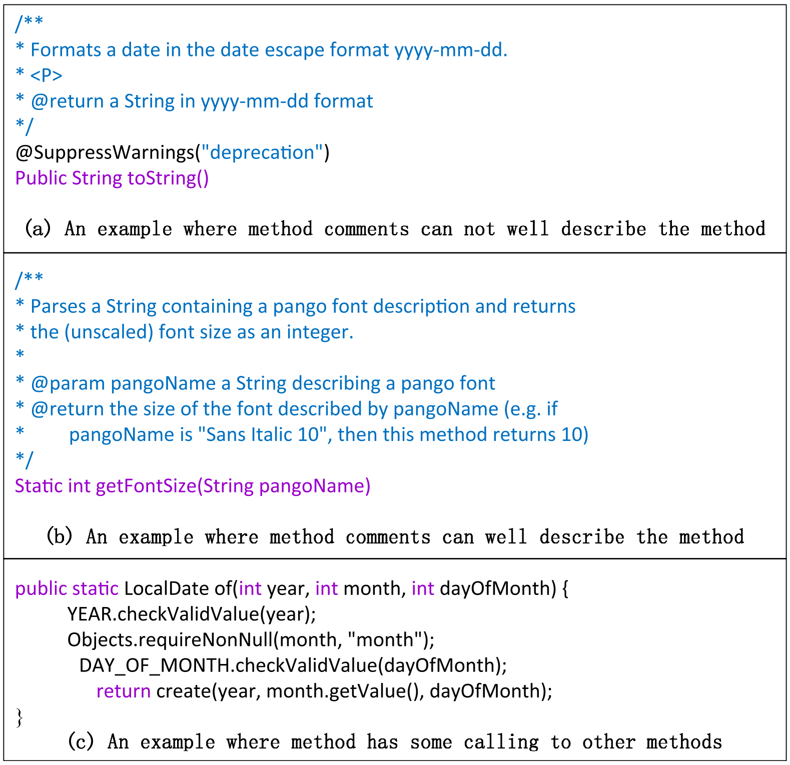
To measure the correlation between method and its comments, we define a metric called *correlation-method*, which evaluates the relevance of the method to its comments. Figure 3(a) shows an example, where the comment does not describe the method well. Figure 3(b) shows an example where the comment well describes the method.

Metric Computation. We extract words contained in the method name and compare them to the words contained in the comments. Then, we count how many words in the method's name are included in the comments. The *correlation-method* denotes the percentage of the number of method's words included in the method comments divided by the total of method's words. Finally, we compare the result with the threshold θ_{method}. The comments in Fig. 3(a) have *correlation-method* $= 0$, and *correlation-method* $= \frac{2}{3}$ in Fig. 3(b).

Based on some experiments with manual evaluation, we set the threshold θ_{method} as 0.5. So, we have,

Hypothesis 5. Comments with *correlation-method* < 0.5 indicate that the correlation between the method and its comments is bad. They should add other description to this method's comments.

Hypothesis 6. Comments with *correlation-method* ≥ 0.5 are assumed to be good in that the correlation between the method and its comments is high.



```
/**
* Formats a date in the date escape format yyyy-mm-dd.
* <P>
* @return a String in yyyy-mm-dd format
*/
@SuppressWarnings("deprecation")
Public String toString()
```

(a) An example where method comments can not well describe the method

```
/**
* Parses a String containing a pango font description and returns
* the (unscaled) font size as an integer.
*
* @param pangoName a String describing a pango font
* @return the size of the font described by pangoName (e.g. if
*        pangoName is "Sans Italic 10", then this method returns 10)
*/
Static int getFontSize(String pangoName)
```

(b) An example where method comments can well describe the method

```
public static LocalDate of(int year, int month, int dayOfMonth) {
        YEAR.checkValidValue(year);
        Objects.requireNonNull(month, "month");
          DAY_OF_MONTH.checkValidValue(dayOfMonth);
             return create(year, month.getValue(), dayOfMonth);
}
```

(c) An example where method has some calling to other methods

Fig. 3. An example of analyzing the method comment.

2.4.4. *Number of lines of code in a method*

For the method that has no comments, but needs to be commented, we use the lines of code in a method as a metric to analyze, whether this method needs comments.

Metric Computation. We extract the code in the method and count its lines of code. Then, we compare the result with a threshold θ_{line}.

Based on some experiments with manual evaluation, we set the threshold θ_{line} as 30. Thus, we have,

Hypothesis 7. A method with more than 30 lines of code should be commented, because developers often do not understand such long method easily without any comments.

2.4.5. *Number of method invocations in a method*

If a method has at most 30 code lines, it is difficult to judge, whether the method needs a comment or not. To decide whether a comment is needed, we need to investigate the complexity of the method. In this paper, we consider a simple way to measure the method complexity: if a method calls many other methods, then we deem a method to be fairly complex. If a method includes only a few lines of code, but these few lines of code are mostly invocations of other methods, the method becomes more difficult to understand. Figure 3(c) shows an example of a method which invokes many other methods in its body.

Metric Computation. We extract the number of method invocations within a method body. Then, we compare the result with a threshold θ_{call}.

Based on some experiments with manual evaluation, we set the threshold θ_{call} as 3. Thus, we have,

Hypothesis 8. A method with more than three method invocations in its body should be commented, because developers cannot understand this method easily.

3. Study Settings and Design

Our approach aims to analyze and assess code comment quality and give suggestions to help developers improve the comment quality. Based on this general objective, we formulate the following research questions (RQs) that will be answered in our user study:

RQ1: Are the metrics proposed in our approach reasonable to distinguish the comments of different-level quality?
RQ2: Are the assessments made by our approach reasonable to reflect the comment quality?
RQ3: Are the suggestions provided by our approach useful for improving comment quality?
RQ4: Can our approach improve over the state-of-art comment quality assessment approaches?

3.1. *Training data*

In our study, we selected code files from two open source projects such as jdk8.0[c] and jEdit.[d] To evaluate our approach, we need to manually select a training data with two types of comments, high quality and low quality. To achieve this, we asked the participants to select some code files, where the comments have high quality or low quality that should be improved. Only the code files that at least 80% of them reached a consensus on the quality results of the comments are included in our study. Finally, we have 50 code files with good comment quality and 30 code files with low comment quality. Among the code files with high quality, 30 code files are from jdk8.0, while the others are from JEdit. In addition, among the codes files of low quality, 10 code files are from jdk8.0, and 20 code files are form JEdit.

[c]http://www.oracle.com/technetwork/java/javase/downloads/index.html.
[d]http://sourceforge.net/projects/jedit/?source=directory.

3.2. *Participants*

In our study, there are 20 volunteer participants with varying levels of programming skills and experiences. Table 1 shows characteristics of the participants. In addition, among these participants, three have industry experience. For RQ1, participants were asked to participate in a survey with a number of multiple choice questions. For RQ2, RQ3 and RQ4, we asked the participants to assess comment quality of a number of header comments and method comments. They form their assessment based on the quality of the comments. For example, if they can easily understand a code by reading a given comment, they will mark the comment to be of high quality.

3.3. *Design*

For RQ1, we asked all participants to answer some questions to evaluate the rationality of the proposed metrics. First, we selected code files according to different metrics. Second, we asked participants to answer some multiple choice questions in a survey. Then, we calculated the ratios of participants that selected various answers. Based on these ratios, we evaluate if a hypothesis is supported by the participants or not.

For RQ2, we randomly selected 20 code files from jdk8.0 and 15 code files from jEdit. Then, we asked the participants to assess the comment quality of these code files, and we also used our approach to assess the comment quality. Next, we compared the assessment results that were generated by our approach with the manual assessment results made by the participants to evaluate whether our assessments were reasonable.

For RQ3, first, we selected the code files of low quality judged in RQ2. There are in total 50 header and method comments of low quality assessed by our approach. Then, the second author of this paper revised the header and method comments based on the 50 suggestions provided by our approach. Finally, we asked the participants to compare the original header and method comments with corresponding revised ones. If participants considered that revised comments are better than unrevised comments, they should vote 1. Otherwise if they considered the revised comments are worse or the original comments do not need to be revised, they vote 0. We calculated the percentage of the cases for which the participant votes are 1.

For RQ4, we compared our approach with Steidl *et al.*'s approach [8] and Liu *et al.*'s approach [12] to see whether our approach improves their effectiveness. Steidl *et al.* proposed to use coherence between code and comments (c_coeff) to measure comment quality. Their approach analyzed the quality of method comments but not

Table 1. Participants.

Education level	Years of programming	Number of developers
Graduate	≥ 5	4
Undergraduate	2	16

class comments, and it can *only identify comments of bad quality.* Comments with c_coeff $= 0$ or c_coeff > 0.5 are deemed as bad comments. They "assume the middle group ($0 <$ c_coeff ≤ 0.5) to be a gray area" with some comments being bad and other comments being good [8]. Liu *et al.* analyzed program readability by using WordNet's synonym matching capability to handle the semantic ambiguity problem in existing studies [12]. Their focus is to find a matched words pair from the code and comments based on their verbs and nouns. To fairly compare two approaches with our approach (which identifies both bad and good comments), we only used methods whose comments are of bad quality to evaluate how good each of the two approaches is. We randomly selected 30 method comments of bad quality (as judged by the participants), and used these two approaches to assess the quality of these comments. Finally, we compared the two approaches' accuracy in identifying bad method comments.

4. Study Results

In this section, we give and discuss the study results.

4.1. *Research question 1*

The purpose of RQ1 is to see whether the metrics in our approach are reasonable. To answer RQ1, for each metric, we asked participants a multiple choice question and showed them some sample comments. In this subsection, for each metric, we describe a multiple choice question and elaborate the process, that we follow to generate the sample comments. We also present the results of our user study.

4.1.1. *Evaluation of the existence of authorship metric*

In order to evaluate the existence of authorship metric, our survey included the following question:

Survey Question 1. Please decide for the header comments in each code file:

(a) The comments have no description of the authorship information, but they need such information.
(b) The comments have no description of the authorship information, and they do not need such information.
(c) The comments have the description of the authorship information, and the authorship information in the comments is important.
(d) The comments have the description of the authorship information, but they do not need such information.

Based on this survey, we can know whether this metric is important. *Hypothesis* 1 is supported if developers vote for the answer (a). Or if developers vote for the answer (c), *Hypothesis* 2 is supported.

Sampling. To evaluate the two hypotheses (*Hypothesis* 1 and *Hypothesis* 2), we sampled header comments according to the existence of the authorship information. We divided the comments into two categories based on whether they have the authorship information. From each category, we randomly sampled 10 comments.

Result. The results of the survey show that for header comments with authorship information, all participants voted for (a). They considered the authorship information in header comments as important and necessary, and every code file should have such information. For header comments without any authorship information, all participants voted for (c). They also considered header comments with authorship information are good. So this survey result supported *Hypothesis* 1.

Implications. After this survey, we consider the *existence-authorship* metric is a useful metric. For the header comments without this information, the developers should add authorship description to this code file.

4.1.2. *Evaluation of correlation metric*

For Header Comments

In order to evaluate the correlation between the header comments and the class, we asked developers the following survey question.

Survey Question 2. Please decide for each comment:

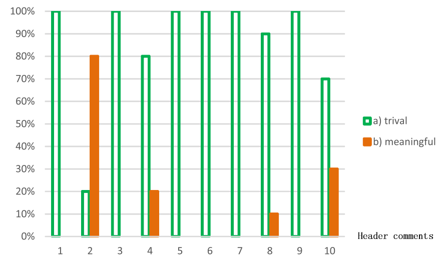
(a) The header comments provide useful information about this class, and the comments are meaningful.
(b) The header comments provide information irrelevant to this class, and the comments should add some additional information to describe this class.

For the header comments with *correlation-headerclass* < 0.5, if participants vote for answer (b), *Hypothesis* 3 is supported. Otherwise, for *correlation-headerclass* ≥ 0.5, if participants vote for answer (a), *Hypothesis* 4 is supported.

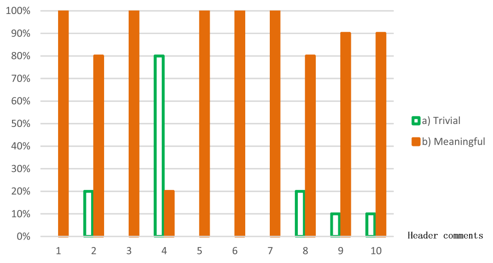
Sampling. To evaluate these two hypotheses (*Hypothesis* 3 and *Hypothesis* 4), we sampled header comments according to the correlation between the header comments and the class, one group with *correlation-headerclass* < 0.5, the other with *correlation-headerclass* ≥ 0.5. Then, we randomly sampled ten header comments for each group.

Result. For header comments with *correlation-headerclass* < 0.5, as Fig. 4(a) shows, eight of 10 participants vote for answer (b), they consider these header comments have little relation to the class, and the comments do not provide useful information about the class. They think that these header comments should add some additional information. Figure 4(b) shows the results of the survey for header comments with *correlation-headerclass* ≥ 0.5. The results show that 85% of the participants consider the header comments are relevant to the class, and the comments provide useful information about this class.

Implications. Based on the above survey, we consider the *correlation-headerclass* metric is a useful metric to detect the relation between header comments and



(a)



(b)

Fig. 4. Survey results for comments with different correlation-headerclass values. (a) The results of votes for correlation–headerclass < 0.5 and (b) The results of votes correlation–headerclass $>= 0.5$.

the class. For header comments with high *correlation-headerclass*, the developers can easily understand the class's functionality. But for header comments with low *correlation-headerclass*, the header comments should add additional information to describe the class to help developers understand it.

For Existing Method Comments

In order to evaluate the correlation between the method and its comments, we asked developers the following survey question.

Survey Question 3. Please decide for each method comments:

(a) The comments provide useful information about this method, and they are meaningful.
(b) The comments provide information irrelevant to this method, and they need to add some additional information about this method's functionality and so on.

For comments with *correlation-method* < 0.5, if participants vote for answer (b), *Hypothesis* 5 is supported. Otherwise, for comments with *correlation-method* ≥ 0.5, if participants vote for answer (a), *Hypothesis* 6 is supported.

Sampling. To evaluate these hypotheses (*Hypothesis* 5 and *Hypothesis* 6), we sampled method comments according to the correlation between comments and the methods, one group with *correlation-method* < 0.5, the other with *correlation-method* ≥ 0.5. We randomly sampled 10 method comments for each group.

Result. Figure 5(a) shows the survey results for comments with *correlation-method* < 0.5. The results show that 79% of the participants consider these comments are trivial, and they do not provide useful information about their functionality, so they vote for (a). For comments with *correlation-method* ≥ 0.5, 91% of the participants vote for answer (b), they consider these comments are sufficient and provide useful information about the methods, as Fig. 5(b) shows.

Implications. After the survey of evaluating the *correlation-method* metric, we consider this metric as a useful metric to detect the relation between the method and its comments. For comments irrelevant to the method, the developers should add some additional information to this method.

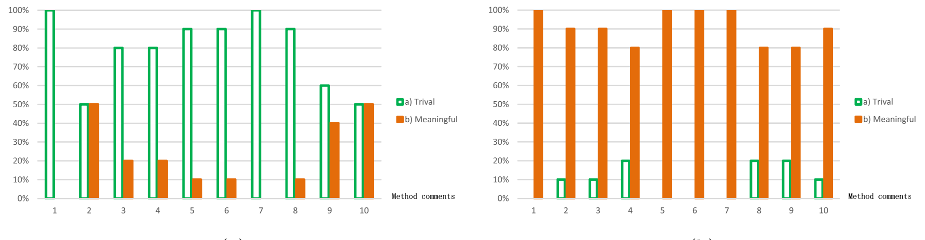


Fig. 5. Survey results for comments with different correlation-method values. (a) The results of votes for correlation–method < 0.5 and (b) The results of votes for correlation–method $>= 0.5$.

4.1.3. *Evaluation of the number of lines of code in a method*

In order to evaluate the rationale of using the number of lines of code in a method to assess the need for code comments, we asked the participants the following survey question:

Survey Question 4. For the method without any comments, please decide:

(a) This method needs comments to make developers easily understand it.
(b) This method is easy to understand, and can have no comment.

Based on this survey question, *Hypothesis* 7 is supported if participants vote for answer (a).

Sampling. To evaluate the hypothesis, we sampled method comments according to the number of lines of code in the method's body. We group them into two groups based on the threshold 30: one with at most 30 code lines, another with more than 30 code lines. For each group we randomly sampled 10 comments.

Result. As Fig. 6(a) shows, for methods with more than 30 lines of code, 66% of the participants consider they need comments to help developers easily understand them. Figure 6(b) shows the results of methods with at most 30 lines of code. Some participants think these methods do not need comments, while some others consider they need comments. Thus, for these methods, the number of lines of code alone is often not sufficient to decide whether code comments are needed or not.

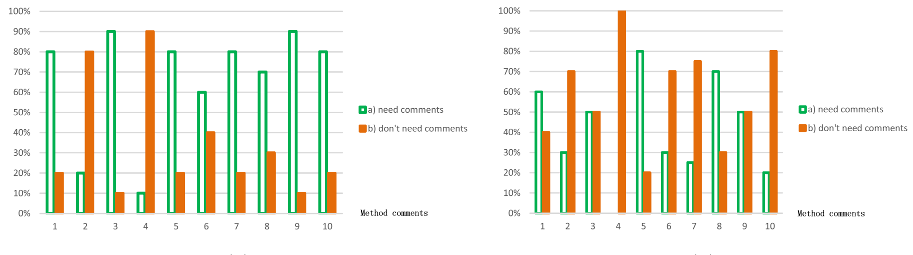


Fig. 6. Survey results for comments with at least or at most 30 lines of code. (a) The results of votes for more than 30 lines of code and (b) The results of votes for at most 30 lines of code.

Implications. After the survey of evaluating the method's lines of code metric, we consider this metric as a useful metric to analyze if the method needs to be commented. For methods with more than 30 lines of code, about 66% of the participants consider they need comments, but for methods with at most 30 lines of code, we cannot decide if this method needs comments, so we need to consider another piece of information.

4.1.4. *Evaluation of the number of method invocations in a method body*

In order to evaluate the number of calling other methods, based on the above survey, we asked the participants the following survey question:

Survey Question 5. For methods with at most 30 lines of code, please decide:

(a) This method needs comments to make developers easily understand it.
(b) This method is easy for developers to understand, and can have no comment.

For methods with at most 30 lines of code, if participants vote for answer (a), *Hypothesis* 8 is supported.

Sampling. We sampled methods according to the number of method invocations in a method body. We divide them into two groups, one group with at most 3 calling other methods, the other with more than 3. For each group, we randomly sampled 10 comments for the study.

Result. As Fig. 7(a) shows, for three out of 10 methods that call at most three methods, developers consider they need comments and vote for answer (a), and some people think other methods do not need comments. For some methods calling more than three methods, Fig. 7(b) shows that about 85% of the participants vote for answer (a), they consider these methods should add comments.

Implications. Through the survey of evaluating the number of method invocations in a method body, the number of calling other methods metric is useful to analyze if the method needs to be commented. For methods calling at most three methods, the number of voting for answer (a) is the same as voting for (b). For

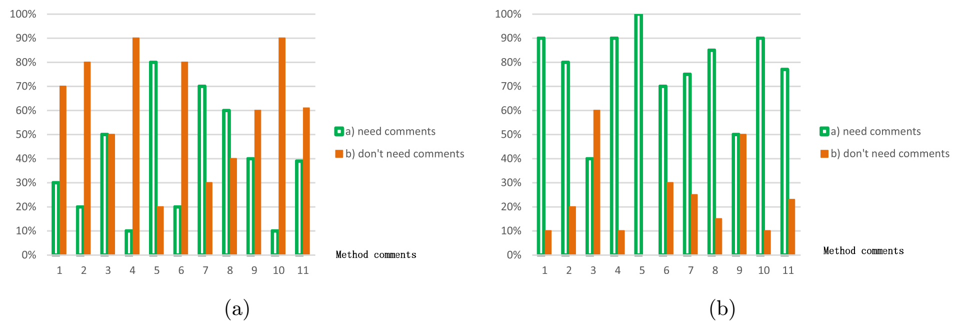


Fig. 7. Survey results for comments calling at most or at least three methods. (a) The results of votes for calling at most 3 methods and (b) The results of votes for calling more than 3 methods.

methods calling more than three methods, this metric is useful to analyze if this method needs comments.

4.2. *Research question 2*

We compared the assessment results generated by our approach with the manual assessment results made by the participants. The results show that 80% of the files are correctly judged by our approach (i.e. its assessments are the same as the participants' assessments). Among these, for code files with good quality, there are about 90% of files that are correctly judged, and about 70% of files with poor quality are correctly judged by our approach. Hence, we can conclude that the automated assessment of the quality of comments in the files, which are made by our approach, are reasonable.

4.3. *Research question 3*

In this study, we focus on whether the suggestions provided by our approach can improve the comment quality. As introduced in Sec. 3.3, we used the suggestions provided by our approach to revise the comments judged as low quality. Then, they were evaluated again by the participants.

The results show that, for 50 comments judged as low quality, developers consider that about 80% of the revised comments become better than their original ones. This indicates that the suggestions provided by our approach can effectively improve the quality of the comments with low quality. Although some suggestions do not improve the quality of the comments, most suggestions are useful for developers to revise the comments.

4.4. *Research question 4*

For RQ4, we would like to know whether our approach improves the effectiveness of the state-of-art approaches on comment quality analysis. In our study, we select two recently proposed work by Steidl *et al.* [8] and Liu *et al.* [12] to compare against each other. Since Steidl *et al.*'s approach can only identify bad method comments, we use 30 methods with bad quality comments to evaluate our and Steidl *et al.*'s approach. The results are shown in Table 2. The results show that for 93% of the methods (i.e. 28 out of the 30 methods), our approach's assessments are correct. On the other hand, Steidl *et al.*'s approach is only correct for 86% of the methods (i.e. 26 out of the

Table 2. The percentage of the methods correctly assessed.

Steidl *et al.*'s approach	86%
Liu *et al.*'s approach	53%
Our approach	93%

30 methods), Liu *et al.*'s approach is only correct for 53% of the methods (i.e. 16 out of the 30 methods). The results show that our approach can improve the accuracy of the state-of-art approaches in detecting poor quality comments.

4.5. *Threats to validity*

Like any empirical validation, our study has its limitations. In the following, some threats to the validity of our empirical studies are discussed.

The main threat to validity comes from the participants. Most of the participants in our study come from university (only three from industry) and have limited developing experiences. Hence, they hold different views about the comment quality. Some of them do not have enough programming experience and do not understand the functionality of comments. To address this threat, we first list the functions of header and method comments. Participants need to read these descriptions. We also used some examples to illustrate good comments and bad comments. Then, participants perform the study and vote for questions according to their understanding and experience. Students mimic newcomers to a software project, who are likely to read comments and benefit more from these comments. Past studies have also highlighted the value of using students as participants of a software engineering user study, e.g. [13].

5. Related Work

A large amount of work has been devoted to software quality analysis or program quality analysis [3, 12, 14], but few on comment quality analysis. In this section, we discuss some related work that only investigated the comment quality.

Steidl and Hummel focused on analyzing member and inline comments [8]. First, they defined seven different comment categories. Then, they proposed two metrics, coherence between code and comments (c-coeff) for member comments and the length of comments for inline comments to measure their quality. Their work assessed the code with comments, but for code without comments, their approach is not suitable. Our work differs from their work. In this paper, we do not analyze the inline comments; instead, we provide an assessment of comment quality on the header and method comments. In addition, we focus on code with comments as well as those without comments. Moreover, some metrics are proposed to assess the quality of the whole comments' quality in a source code file.

Liu *et al.* proposed an approach to analyze program readability based on WordNet, which is able to expand the range of keyword search and solve the problem of semantic ambiguity for natural language comment analysis [12]. They mainly focused on identifying the inconsistency between the methods and their corresponding comments in the program. In this paper, we provided an comprehensive assessment of comment quality on the header and method comments. In addition, we not only analyze the code with comments, but also those without comments.

Aman *et al.* studied the impacts of comment statements on source code stability in an open source development [15]. The results show that describing many comments is recommend for developing a stable code. So, in this paper, we also focus on the code that does need the comments, which can help improve the stability of the source code.

Khamis *et al.* proposed a tool, *JavadocMiner*, to analyze the quality of *Javadoc* comments [7]. They aimed at evaluating the quality of language used in comments and the consistency between source code and comments. They targeted the same research question as we do, that is how to measure the comment quality. But we analyzed the comment quality in the program, which is different from theirs.

Lawrie *et al.* used information retrieval techniques to assess the function quality under the assumption that "if the code is of high quality, the comments will give a good description of the source code" [16]. Similar to our work, we also investigated the similarity relation between source code and comments. However, we compared the relation between comments and class name, method name, which were ignored in their work.

6. Conclusion

This paper proposed an approach to automatically analyze and assess the quality of the header and member method comments in a program. For header comments, we used two metrics: the existence of authorship information and the correlation between the header comments and the class. For member method comments, we had three metrics: the correlation between the method and its comments, the number of lines of code in a method body, and the number of method invocations in a method body. After analyzing these comments, our approach provides some suggestions to revise header and member method comments to improve their quality. Finally, our approach assessed the whole comments' quality based on the ratio of the comments' suggestions. The empirical studies were conducted on some source code files from jdk8.0 and jEdit. The results show that our approach can provide reasonable comment quality analysis results and useful suggestions for improved comment quality. In addition, the comparative study with the state-of-art approaches shows that the accuracy of their comment quality analysis on poor method comments can be improved.

In our approach, we only analyzed the comment quality based on the lexicon of the source code and the comments such as the correlation between comments and code. We did not take their semantics into consideration. For example, if a method declares the sum of two numbers and the method's name is *add*, but the comments may use another word such as *sum*. Although the used word in the method's name is different in its comments, their semantics are similar. In our future work, we will consider their semantics to improve the accuracy of comment quality assessment. Finally, this paper only analyzed two types of comments. There are also some other types of comments, for example, inline comments, task comments, etc. We will analyze the quality of these types of comments in our future work.

Acknowledgments

This work is supported partially by Natural Science Foundation of China under Grant Nos. 61402396 and 61472344, partially by the Natural Science Foundation of the Jiangsu Higher Education Institutions of China under Grant No. 13KJB520027, the China Postdoctoral Science Foundation under Grant No. 2015M571489, and partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant No. KFKT20146B21. A special thank to the anonymous reviewers who provided useful suggestions on the earlier version of this paper.

References

1. A. von Mayrhauser and A. M. Vans, Program comprehension during software maintenance and evolutions, *IEEE Comput.* **28**(8) (1995) 44–55.
2. E. Soloway and K. Ehrlich, Empirical studies of programming knowledge, *IEEE Trans. Softw. Eng.* **10**(5) (1984) 595–609.
3. T. M. Khoshgoftaar, Y. Xiao and K. Gao, Software quality assessment using a multi-strategy classifier, *Inf. Sci.* **259** (2014) 555–570.
4. I. Stamelos, L. Angelis, A. Oikonomou and G. L. Bleris, Code quality analysis in open source software development, *Inf. Syst. J.* **12**(1) (2002) 43–60.
5. K. Gao, T. M. Khoshgoftaar and N. Seliya, Predicting high-risk program modules by selecting the right software measurements, *Softw. Qual. J.* **20**(1) (2012) 3–42.
6. S. Barney, V. Mohankumar, P. Chatzipetrou, A. Aurum, C. Wohlin and L. Angelis, Software quality across borders: Three case studies on company internal alignment, *Inf. Softw. Technol.* **56**(1) (2014) 20–38.
7. N. Khamis, R. Witte and J. Rilling, Automatic quality assessment of source code comments: The JavadocMiner, in *NLDB*, Lecture Notes in Computer Science Vol. 6177 (Springer, 2010), pp. 68–79.
8. D. Steidl, B. Hummel and E. Jürgens, Quality analysis of source code comments, in *IEEE 21st Int. Conf. Program Comprehension*, 20–21 May 2013, San Francisco, CA, USA, pp. 83–92.
9. S. N. Woodfield, H. E. Dunsmore and V. Y. Shen, The effect of modularization and comments on program comprehension, in *Proc 5th Int. Conf. Software Engineering,* 9–12 March, 1981. San Diego, California, USA, pp. 215–223.
10. A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur and P. Thompson, *The Elements of Java Style* (Cambridge University Press, Cambridge, 2000).
11. M. L. Vásquez, K. Hossen, H. Dang, H. H. Kagdi, M. Gethers and D. Poshyvanyk, Triaging incoming change requests: Bug or commit history, or code authorship? in *28th IEEE Int. Conf. Software Maintenance*, Trento, Italy, September 23–28, 2012, pp. 451–460.
12. Y. Liu, X. Sun and Y. Duan, Analyzing program readability based on WordNet, in *Proc. 19th Int. Conf. Evaluation and Assessment in Software Engineering* (ACM, New York, 2015), pp. 27:1–27:2.
13. M. Höst, B. Regnell and C. Wohlin, Using students as subjects — A comparative study of students and professionals in lead-time impact assessment, *Empir. Softw. Eng.* **5**(3) (2000) 201–214.
14. T. M. Khoshgoftaar, P. Rebours and N. Seliya, Software quality analysis by combining multiple projects and learners, *Softw. Qual. J.* **17**(1) (2009) 25–49.

15. H. Aman and H. Okazaki, Impact of comment statement on code stability in open source development, in *Proc. Conf. Knowledge-Based Software Engineering*, 2008, pp. 415–419.
16. D. J. Lawrie, H. Feild and D. Binkley, Leveraged quality assessment using information retrieval techniques, in *14th Int. Conf. Program Comprehension*, 14–16 June 2006, Athens, Greece, pp. 149–158.